Department of Public Works

September 6th, 2011

San Gabriel Valley Pilot Project

INFORMATION EXCHANGE NETWORK

Recommendations
for the implementation of
new Traffic Control System
Command/Data Interface Programs

Final, Revision 6
2nd Submittal

METRO

PUBLIC WORKS

Prepared by:

TRANSCORE

626 Wilshire Blvd. Suite 818
Los Angeles, CA 90017

LOS ANGELES COUNTY

INFORMATION EXCHANGE NETWORK (IEN)

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

FINAL – REVISION 6

(2nd Submittal)

Prepared for:

Los Angeles County
Department of Public Works

Prepared by:

TRANSCORE

626 Wilshire Boulevard
Suite 818
Los Angeles, CA 90017

September 6th, 2011

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	1/20/2005	• Update Section 2.4 TCS CDI Event Format according to IEN implementation
Final – Revision 4	1/03/2006	• Series 2000 CDI Software Modifications
Final – Revision 5	5/19/2006	• Minor changes to detector status field descriptions
Final – Revision 6 (1st Submittal) Final – Revision 6 (2nd Submittal)	6/29/2009 9/6/2011	• Changes for the IEN Multiple Corridor Server (MCS) Project – Version 3 of the CDI • Addresses minor LA County comments

1. INTRODUCTION

1.1 PURPOSE OF THE DOCUMENT

The Los Angeles County Information Exchange Network (IEN) uses command/data interface (CDI) programs to connect 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 includes an example CDI based on the CDI that TransCore wrote for its own *TransSuite*® TCS.

1.2 CDI INTERFACE HISTORY

The IEN has used three versions of the command/data interface. Version 1 was used in the San Gabriel Valley Pilot Project and only communicated with TransCore's Series 2000 TCS. Version 2 was created for TCS vendors other than TransCore, and was the version documented in all revisions of this document before Revision 6. Version 3 of the CDI has been created for the IEN Multiple Corridor Server (MCS) Project.

1.3 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.

1.4 ASSUMPTIONS

The IEN Site Server application communicates with TCS CDI programs. It uses the JacORB 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 Department of Public Works (LA County) holds the copyright to the source code developed by TransCore specifically for the IEN program. LA County may grant developers of TCS CDI programs the right to view or copy the source code of the sample CDI shown in this document, which may be used as a guide to development of other CDIs.

2. CHANGING FROM VERSION 2 TO VERSION 3

This section describes the changes between Version 2 of the CDI and Version 3 and how to modify an existing Version 2 interface program to support Version 3.

2.1 NEW INTERFACES

In Version 3 of the CDI, new *MCSDataAccessor* and *MCSCommandAccessor* interfaces have been derived from the *DataAccessor* and *CommandAccessor* interfaces from Version 2 of the CDI. The old interfaces are still defined in the IDL because the IEN Site Server process must support both Versions of the interface.

2.1.1 Explicit Interface Definition in IDL

The *MCSDataAccessor* and *MCSCommandAccessor* interfaces explicitly define all parts of the interface formerly governed by convention. The new interfaces use IDL enumerations and structures to define device codes and data structures received from and sent to the CDI rather than integers and sequences of integers. This change should simplify the process of providing device status data for the IEN Site Server, which is the most important task of the CDI.

2.1.2 Increased Device Identifier Size

All IEN device identifiers have been increased in size from 16 bits to 32 bits. Several traffic control systems use device identifiers larger than 65,535, the former maximum size for device identifiers. New methods have been defined in the *MCSDataAccessor* and *MCSCommandAccessor* interfaces that accept 32-bit device identifiers.

2.1.3 Additional Intersection Configuration Data

Version 3 of the CDI requests more intersection configuration data from the traffic control system. See Table 2-1 below for a list of new intersection configuration data requested.

Table 2-1: Additional Version 3 Intersection Configuration Data

NEW FIELD	USAGE
mainStreet	Name of main street for the intersection.
crossStreet	Name of cross street for the intersection.
mainStreetDirection	Direction of traffic flow on the main street.
latitude	Latitude of the intersection in microdegrees.
longitude	Longitude of the intersection in microdegrees.

2.1.4 Individual Device Error Return Codes

In Version 3 of the CDI, the data request and command methods of the CDI can return error codes for each device for which access failed. The corresponding methods in Version 2 of the CDI could indicate an error, but not the particular device which caused the error.

2.2 MODIFYING AN EXISTING CDI PROGRAM

This section describes the sections of a Version 2 CDI that would have to be changed to support Version 3 of the interface definition.

2.2.1 Changes to DataAccessorFactory and CommandAccessorFactory Objects

The IEN Site Server calls the CDI's *DataAccessorFactory* object to get an instance of a *DataAccessor* object from which to request CDI data. The `createDataAccessor` method of the *DataAccessorFactory* object exposed by a Version 3 CDI should return a reference to an *MCSDataAccessor* object.

Similarly, the IEN Site Server calls the CDI's *CommandAccessorFactory* object to get an instance of a *CommandAccessor* with which to send commands to the CDI. The `createCommandAccessor` method of the *CommandAccessorFactory* object exposed by a Version 3 CDI should return a reference to a *MCSCommandAccessor* object.

2.2.2 Changing the DataAccessor Object to Implement MCSDataAccessor

The DataAccessor servant in the CDI must be changed to implement the MCSDataAccessor interface. MCSDataAccessor replaces three methods in the DataAccessor interface with three new methods, as shown in Table 2-2 below.

Table 2-2: Methods Replaced in the MCSDataAccessor Interface

DATAACCESSOR METHOD	MCSDATAACCESSOR METHOD	CHANGES
getDeviceList	getAvailableDevices32	Caller passes in list of device types of interest; CDI returns list with 32-bit device IDs.
deviceDataTypes	getDeviceEventTypes	Requested event types for devices now defined by enumerations instead of integers.
getDeviceEventDataList	getDeviceUpdate32	List of requested devices and data types passed to the CDI contains 32-bit device identifiers and enumerations for requested data types; CDI returns list of errors matched to device ID and data type.

2.2.3 Changing the CommandAccessor Object to Implement MCSCommandAccessor

The *CommandAccessor* servant in the CDI must be changed to implement the *MCSCommandAccessor* interface. *MCSCommandAccessor* adds two new methods and replaces three existing methods of the *DataAccessor* interface, as shown in

Table 2-3 below.

Table 2-3: Methods Changed in the MCSCommandAccessor Interface

COMMANDACCESSOR METHOD	*MCSCOMMANDACCESSOR* METHOD	CHANGES
N/A	getAvailableDevices32	New; returns a list of devices for which the CDI will accept commands.
setCDIPlan	setCDIPlan32	Device list passed to the method contains 32-bit device identifiers; method returns results for each device in the request list.
changeMode	changeMode32	Device list passed to the method contains 32-bit device identifiers; method returns results for each device in the request list.
releaseControl	releaseControl32	Device list passed to the method contains 32-bit device identifiers; method returns results for each device in the request list.
N/A	flash32	New; requests the CDI to put a list of devices into or out of flash.

3. 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 *TransSuite*® TCS, a CDI for a BI Tran QuicNet system, the LA County TCS, or others.

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

TCS CDI programs must implement the objects defined in the CORBA interfaces that are defined in *Section 3.1: 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 3.2: 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 3.3: Usage of the CORBA Naming Service*.
- TCS CDI programs must format data to be transmitted to the IEN as described in *Section 3.1.7.2: Device Event Structures*.

This section of the document begins by describing the nine (9) IDL files that define the IEN interface to TCS CDI programs. The section continues with CDI performance requirements, usage of the CORBA naming service, and considerations for implementing a CDI that must communicate with the Site Server through a firewall.

3.1 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:

- IEN.idl
- IENRTData.idl
- tcs.idl
- IENTCSData.idl
- MCSData.idl
- MCSDataInterface.idl
- tcscommand.idl
- mcscommand.idl

Complete listings of the IDL files are provided in *Appendix A: TCS CDI CORBA IDL Files*.

3.1.1 IEN.idl

The IEN.idl file contains the IEN module. It defines basic sequences, typedefs, and exceptions used in structures and interfaces in other modules.

3.1.2 IENRTData.idl

This file defines the IENRTData CORBA module, which contains basic type definitions. The event data structures in the file are used by CDIs that support Version 2 of the interface. Version 3 CDIs do not use event data structures. The *DeviceType* enumeration, which defines the kinds of traffic control system devices used in the IEN, is the only item used from this file:

```
enum DeviceType

{

       DT_SYSTEM,

       DT_SCHEDULE,

       DT_INTERSECTION,

       DT_SECTION,

       DT_DETECTOR,

       DT_SIGN,

       DT_CAMERA,

       DT_HAR,

       DT_DETECTOR_STATION

};
```

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

3.1.3 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.

3.1.3.1 Mode Enumeration

```
/// Modes of operation

    enum Mode

    {

       NORMAL,

       LOCAL_TOD,

       FREE,
```

```
    TOD,

    RESPONSIVE,

    MANUAL,

    RELEASE

};
```

This enumeration defines the control modes that a CDI is required to accept in a command to change a section or intersection's plan selection mode. Table 3-1 below contains an explanation of the enumeration values.

Table 3-1: Commanded 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	This control mode is deprecated. The IEN Site Server program will use the releaseControl32 method of the MCSCommandInterface when releasing control of a device controlled by a CDI running Version 3 of the interface.

Note that the control modes used in commands shown in Table 3-1 above are different from the control modes that the CDI may report, which are listed in Section 3.1.4.2 below.

3.1.3.2 *MCSDevice* and *Device* Structures

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

```
///NEW: 32 bit device identifier instead of 16 bit.

typedef long MCSDeviceID;


///NEW: 32 bit device identifier instead of 16 bit.
```

```
///System ID moved into SiteUpdate in MCSDataInterface.idl.

struct MCSDevice

{

	IENRTData::DeviceType type;

	MCSDeviceID id;

};
```

The *Device* structure was used as the device identifier structure in previous versions of the CDI, but is now deprecated because it does not support device IDs greater than 32,767.

3.1.3.3 *MCSDeviceList* and *DeviceList* Sequences

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

```
	///NEW: 32 bit device identifier instead of 16 bit.

	typedef sequence<MCSDevice> MCSDeviceList
```

The *DeviceList* sequence was used to request a list of available devices in previous versions of the CDI, but is now deprecated because it does not support device IDs greater than 32,767.

3.1.3.4 *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;

};
```

3.1.3.5 *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

};
```

3.1.3.6 *ConfigurationAccessor* Interface

This is a base interface for the *MCSCommandAccessor* interface, which the IEN uses to send commands to a TCS, and the *MCSDataAccessor* interface, 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);
};
```

3.1.4 IENTCSData.idl

The `IENTCSData.idl` file defines the TCSData module. It depends on the `TCS.`idl and `IENRtData.idl` files. It defines the *DataAccessor* interface, which is the base of the *MCSDataAccessor* interface that the IEN Site Server program will call to get traffic control system data, and the *DataAccessorFactory* interface, which the IEN Site Server calls to get an instance of the *MCSDataAccessor*. It also defines several enumerated types that are used in the new *MCSDataAccessor* interface.

The *DataAccessor* interface in the `IENTCSData.idl` file is the interface that the IEN Site Server uses to get data from a CDI that supports Version 2 of the interface. The IEN Site Server will use the new *MCSDataAccessor* interface to get data from a Version 3 CDI. The *MCSDataAccessor* interface is derived from the *DataAccessor* interface. All new CDI implementations should support the new *MCSDataAccessor* interface.

3.1.4.1 *IEN_EventType* Enumeration

The TCS data interface uses the *IEN_EventType* enumeration for the types of device data requested by the IEN and returned by the CDI. These event types correspond to different structures that should be returned by the CDI when the Site Server calls its getDeviceUpdate32 method. The structures themselves are defined in Section 3.1.7.2 below.

```
        enum IEN_EventType
        {
            IEN_COMMANDRETURN,
            IEN_INTERSECTIONRTSTATUS,
            IEN_INTERSECTIONRTSUMMARY,
```

```
        IEN_PHASE_STATEDATA,
        IEN_PEDPHASE_STATEDATA,
        IEN_VEHCALL_STATEDATA,
        IEN_LASTCYCLE_PHASEDATA,
        IEN_DETECTORSTATE,
        IEN_INTERSECTIONINFO,
        IEN_DETECTORINFO,
        IEN_TP_PHASEDATA,
        IEN_SECTIONINFO,
        IEN_SECTIONSTATE
    };
```

Table 3-2 below shows the data structures defined in Section 3.1.7.2 for each IEN event.

Table 3-2: IEN Event Types and Data Structures

EVENT TYPE	DATA STRUCTURE	DOCUMENT SECTION
IEN_COMMANDRETURN	Not used	N/A
IEN_INTERSECTIONRTSTATUS	IntersectionRTStatus	3.1.7.2.3
IEN_INTERSECTIONRTSUMMARY	IntersectionRTSummary	3.1.7.2.2
IEN_PHASE_STATEDATA	PhaseStateData	3.1.7.2.4
IEN_PEDPHASE_STATEDATA	PedestrianPhaseState	3.1.7.2.5
IEN_VEHCALL_STATEDATA	VehicleCallState	3.1.7.2.6
IEN_LASTCYCLE_PHASEDATA	LastCyclePhaseData	3.1.7.2.8
IEN_DETECTORSTATE	DetectorState	3.1.7.2.11
IEN_INTERSECTIONINFO	IntersectionInfo	3.1.7.2.1
IEN_DETECTORINFO	DetectorInfo	3.1.7.2.10
IEN_TP_PHASEDATA	TpPhaseData	3.1.7.2.9
IEN_SECTIONINFO	SectionInfo	3.1.7.2.12
IEN_SECTIONSTATE	SectionState	3.1.7.2.13

3.1.4.2 *SignalControlMode* Enumeration

The *SignalControlMode* enumeration describes the control modes that may be reported for intersection controllers. Control modes are used in the IEN to indicate the mode of plan selection used by the controller.

```
enum SignalControlMode

{

    ISC_OTHER_NO_ADDITIONAL,            // A mode other than those here

    ISC_OTHER_ADDITIONAL,               // Deprecated

    ISC_FREE,                           // free

    ISC_FIXED_TIME,                     // Fixed length phases

    ISC_TIME_BASE_COORDINATION,         // Coordinated clock-based plan

    ISC_ACTUATED,                       // Fully actuated ctrl (like free)

    ISC_SEMI_ACTUATED,                  // Semi-actuated (redundant with

                                        // time based coordination,

                                        // deprecated)

    ISC_CRITICAL_INTERSECTION_CONTROL,  // Split adjustment based on traffic

                                        // traffic at a critical intersection

    ISC_TRAFFIC_RESPONSIVE,             // Traffic responsive plan selection

    ISC_ADAPTIVE,                       // Using an adaptive algorithm

    ISC_TRANSITION,                     // Transition between plans

    ISC_EXTERNAL,                       // Plan from external system

    ISC_ATCS,                           // Special LADOT adaptive mode

    ISC_UNKNOWN

};
```

3.1.4.3 *IntersectionSignalState* Enumeration

The *IntersectionSignalState* enumeration denotes the intersection's operational state.

```
enum IntersectionSignalState

{

    ISS_OTHER_NO_ADDITIONAL,            // Obsolete, do not use

    ISS_OTHER_ADDITIONAL,               // Obsolete, do not use

    NORMAL_OPERATION,                   // Intersection operating normally
```

```
        FLASH,                          // Intersection is flashing
        PREEMPTION,                     // Preemption input active
        CONFLICT_FLASH,                 // Flashing due to conflict monitor
        FAILED,                         // Central system has failed the
                                        // controller
        ISS_UNKNOWN                     // State unknown
    };
```

3.1.4.4 *ControllerResponseState* Enumeration

The *ControllerResponseState* indicates whether or not the controller responded to the most recent communication attempts. The `RESPONDING`, `NOT_RESPONDING`, and `UNKNOWN` values should be used for most intersection response states.

```
    enum ControllerResponseState
    {
        RESPONDING_OTHER_NO_ADDITIONAL,     // Obsolete, do not use
        RESPONDING_OTHER_ADDITIONAL,        // Obsolete, do not use
        RESPONDING,                         // Responded to last comm attempt
        NOT_RESPONDING,                     // No response to last comm attempt
        UNKNOWN                             // Response state unknown
    };
```

3.1.4.5 *PreemptType* Enumeration

The *PreemptType* enumeration enables the CDI to report the reason why an intersection is operating under preemption.

```
    enum PreemptType
    {
        PREEMPT_OTHER_NO_ADDITIONAL,        // Preempt of type other than those
                                            // in this enumeration
        PREEMPT_OTHER_ADDITIONAL,           // Obsolete, do not use
        NO_PREEMPT,                         // No preemption in effect
        GENERAL_PREEMPT,                    // General preemption
        BRIDGE_PREEMPT,                     // Preemption by bridge
        EV_PREEMPT,                         // Preemption for emergency vehicle
```

```
        LRT_PREEMPT,                    // Preempt for light rail train
        RR_PREEMPT,                     // Railroad preempt
        PREEMPT_UNKNOWN                 // Preempt of unknown type
    };
```

3.1.4.6 *CommState* Enumeration

This enumeration enables a CDI to report whether or not the TCS can communicate with a controller well enough to determine the controller's state. The enumeration describes the central system's assessment of communication to the controller. The obsolete "OTHER" enumeration values duplicate the more descriptive COMM_UNKNOWN enumeration value.

```
    enum CommState
    {
        COMM_OTHER_NO_ADDITIONAL,       // Obsolete, do not use
        COMM_OTHER_ADDITIONAL,          // Obsolete, do not use
        COMM_GOOD,                      // Comm from central to ctrlr good
                                        // enough for central
        COMM_BAD,                       // Comm from central to ctrlr too
                                        // bad to use reliably
        COMM_UNKNOWN                    // Central can't determine comm
                                        // state to ctrlr, due to comm eqpmt
                                        // failure or ctrlr offline
    };
```

3.1.4.7 Controller Alarm Bitmasks

The following constants define alarm bits that may be reported for IEN controllers in the alarms member of the *IntersectionRTSummary* structure, as described in Section 3.1.7.2.2.

```
    const short NO_ALARM                = 0x00;
    const short CONFLICT_FLASH_ALARM    = 0x01;
    const short CABINET_DOOR_OPEN_ALARM = 0x02;
    const short TRANSITION_ALARM        = 0x04;
    const short INTERNAL_ERROR_ALARM    = 0x08;
    const short FLASH_ALARM             = 0x10;
```

3.1.4.8 *DetectorStatus* Enumeration

The *DetectorStatus* enumeration describes the values that may be returned to describe the status of a detector to the IEN. This enumeration is used for the status member of the DetectorState structure, defined in Section 3.1.7.2.11.

```
enum DetectorStatus

{

    /// Other state; no additional details available

    DETECTOR_OTHER_NO_ADDITIONAL,


    /// Other state; additional details available through

    /// device-specific interface

    DETECTOR_OTHER_ADDITIONAL,


    /// Enabled but not working due to hardware or comm failure

    DETECTOR_FAILED,


    /// working

    DETECTOR_OPERATIONAL,


    /// intentionally disabled

    DETECTOR_OFF,


    /// Detector state unknown due to communication problems, system

    /// configuration, or other central system problems.

    DETECTOR_UNKNOWN

};
```

3.1.4.9 *DetectorClass* Enumeration

The *DetectorClass* enumeration describes how a detector is used by the traffic control system. Most detectors of interest to the IEN will be system detectors (DC_SYSTEM). The *DetectorClass* enumeration is used for the *detector*Class member of the *DetectorInfo* structure defined in Section 3.1.7.2.10.

```
/// Usage of the detector in the traffic system.
```

```
enum DetectorClass
{
    DC_OTHER_NO_ADDITIONAL,
    DC_OTHER_ADDITIONAL,
    DC_STOP_BAR,
    DC_SYSTEM,
    DC_PEDESTRIAN,
    DC_ADAPTIVE,
    DC_CALL,
    DC_EXTENSION,
    DC_MAINLINE,
    DC_REVERSIBLE_LANE,
    DC_RAMP_DEMAND,
    DC_RAMP_MERGE,
    DC_RAMP_PASSAGE,
    DC_RAMP_QUEUE,
    DC_UNKNOWN
};
```

3.1.4.10 *DetectorType* Enumeration

The *DetectorType* enumeration describes the implementation technology for detectors connected to traffic control systems. It is used in the *detector*Type member of the *DetectorInfo* structure defined in Section 3.1.7.2.10.

```
enum DetectorType
{
    DT_OTHER_NO_ADDITIONAL,
    DT_OTHER_ADDITIONAL,
    DT_INDUCTIVE_LOOP,
    DT_MAGNETIC,
    DT_MAGNETOMETERS,
    DT_PRESSURE_CELLS,
    DT_MICROWAVE_RADAR,
```

```
        DT_ULTRASONIC,
        DT_VIDEO_IMAGE,
        DT_LASER,
        DT_INFRARED,
        DT_ROAD_TUBE,
        DT_UNKNOWN
    };
```

3.1.4.11 *Direction* Enumeration

The *Direction* enumeration describes the direction of traffic flow on a roadway or past a detector. It is used in the mainStreetDirection field in the *IntersectionInfo* structure defined in Section 3.1.7.2.1and the detectorDirection field in the DetectorInfo structure defined in Section 3.1.7.2.10.

```
    enum Direction {
        EastBound,
        WestBound,
        SouthBound,
        NorthBound,
        SouthEastBound,
        SouthWestBound,
        NorthEastBound,
        NorthWestBound,
        InBound,
        OutBound,
        None,
        East_West,              // Bi-directional east & west
        North_South,            // Bi-directional north & sourth
        NE_SW,                  // Bi-directional northest and southwest
        NW_SE,                  // Bi-directional northwest and southeast
        InBound_and_Outbound,   // Bi-directional in and outbound
        Other                   // When none of the above will do . . .
      };
```

3.1.5 TCSData.idl

The *TCSData* module contains version constants used in Version 2 CDIs, several struct and typedef definitions that are used by Version 2 CDIs, and definitions of the *DataAccessor* and *DataAccessorFactory* interfaces.

3.1.5.1 IEN CDI Version 2 Interface Version Constants

The first items in the *TCSData* module are the interface constants that version 2 CDIs return to identify themselves. The IEN Site Server program treats any CDI that returns its interfaceVersion property with its majorVersion field set to 2 as Version 2 CDIs.

3.1.5.2 *DataAccessor* Interface

The *DataAccessor* interface of the CDI is used by the Site Servers to get data from Version 2 CDIs. The new *MCSDataAccessor* interface (shown in Section 3.1.7.6) used in Version 3 CDIs is derived from *DataAccessor*. It contains replacement methods for `getDeviceList`, `deviceDataTypes`, and `getDeviceEventDataList` that support 32-bit device identifiers. However, a Version 3 CDI must still support the clientName attribute and destroy method defined in the *DataAccessor* interface.

```
/// 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);
  };
```

3.1.5.2.1 clientName Attribute

This attribute returns the instance name passed to the call to the *DataAccessorFactory* object's createAccessor method that created the *DataAccessor* object.

3.1.5.2.2 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.

3.1.5.2.3 getDeviceList Method

(*Note*: the `getDeviceList` method has been superseded in the *MCSDataInterface* defined for the MCS project. CDIs that support *MCSDataInterface* may return an empty list from this method.)

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 3.1.5.3: DataAccessorFactory Interface* below.

3.1.5.2.4 deviceDataTypes Method

(Note: the deviceDataTypes method has been superseded in the *MCSDataInterface* defined for the MCS project. Version 3 CDIs should return an empty list from this method.) This method returns a list of device types supported by the CDI and the data types that the IEN Site Server may request for each supported device type.

3.1.5.2.5 getDeviceEventDataList Method

The `getDeviceEventDataList` method has been superseded in the *MCSDataInterface* defined for the MCS project. CDIs that support *MCSDataInterface* may return an empty list from this 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 return all requested data for the device.

3.1.5.3 DataAccessorFactory Interface

The IEN Site Server calls this interface when it wishes to begin getting intersection data from a traffic control system. Version 3 CDIs should return an object of type *MCSDataAccessor* downcast to *DataAccessor* from this method. The IEN Site Server will determine whether or not a *DataAccessor* can be cast to *MCSDataAccessor* by the version information it returns. If the major version number is 3 or higher, then the Site Server will cast the *DataAccessor* to an instance of *MCSDataAccessor*.

```
/// 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 );
    };
```

3.1.6 MCSData.idl

The MCSDATA module is defined in the `MCSData.idl` file. It defines sequences and structures for representing devices with 32-bit identifiers.

3.1.6.1 MCSDeviceID typedef

This typedef represents a 32-bit device ID.

```
typedef long  MCSDeviceID;
```

3.1.6.2 MCSDeviceIDList Sequence

This sequence contains a list of 32-bit device identifiers.

```
typedef sequence<MCSDeviceID> MCSDeviceIDList;
```

3.1.6.3 MCSDevice Structure

This structure contains a device type field and a 32-bit device identifier.

```
struct MCSDevice
{
        IENRTData::DeviceType type;
        MCSDeviceID id;
};
```

3.1.6.4 MCSDeviceList Sequence

This sequence contains a list of MCSDevice structures.

```
typedef sequence<MCSDevice> MCSDeviceList;
```

3.1.6.5 EventTypeList Sequence

This sequence contains a list of *IEN_EventType* structures for specifying data types to request from a CDI.

```
typedef sequence<IENTCSData::IEN_EventType> EventTypeList;
```

3.1.7 MCSDataInterface.idl

The `MCSDataInterface.idl` file contains the MCSDataInterface module. The module contains definitions of all MCS device event types as structures and the *MCSDataAccessor* interface. *MCSDataAccessor* is derived from the *DataAccessor* interface, which contains the full set of methods for old CDI programs. It adds methods that return device data with 32-bit device identifiers. New CDI implementations should support all methods from the *ConfigurationAccessor* interface, all methods from the *DataAccessor* interface except for `getDeviceList` and `getDeviceEventDataList`, and all methods from the *MCSDataAccessor* interface.

3.1.7.1 CDI Version 3 Interface Version Number Constants

The constants defined in the beginning of the MCSDataInterface module are the major version number, minor version number, and revision number for the currently defined IEN command data interfaces. IEN Site Server programs require that a CDI's interfaceVersion attribute returns a major version number of 3 if it supports the *MCSDataInterface*. The minorVersion and revision fields may be significant in later revisions of the IEN CDI definition.

```
const short majorVersion = 3;

const short minorVersion = 0;

const short revision     = 0;
```

3.1.7.2 Device Event Structures

MCS device events are defined as IDL structures. They define all events that are returned from the CDI to the Site Server. In Version 2 of the CDI interface, the CDI built events from a generic structure, with variable-length sequences. In Version 3, the CDI should instead use the structures defined in the `MCSDataInterface.idl` file.

The structures defined in this section define the various types of events that the IEN Site Server may request from the CDI for intersection, section, and detector data, and that the Site Server may send to the CDI to request changes in intersection and section operation. Each event structure corresponds to one of the values from the *IEN_EventType* enumeration in the `IENTCSData.idl` file.

The events defined in this section are all components of the *SiteUpdate* structure, which the site server requests from the CDI in the `getDeviceUpdate32` method of the *MCSDataAccessor* interface.

3.1.7.2.1 IntersectionInfo Structure

This structure contains configuration information for an intersection controller. The TCS CDI should return an *IntersectionInfo* structure to the IEN Site Server requests one for a valid

intersection. It is denoted by the IEN_INTERSECTIONINFO enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_INTERSECTIONINFO event defined in Version 2 of the interface.

```
struct IntersectionInfo

{

    ///ID of the controller.

    MCSDATA::MCSDeviceID id;


    ///The EntityID of the section to which this intersection belongs. -1 if

    /// it is not a member of a section.

    long     sectionID;


    ///Seconds between poll attempts to the intersection controller.

    short    secondsBetweenPollAttempts;


    ///The name of the cross street of the intersection.

    string   crossStreet;


    ///The name of the main street of the intersection.

    string   mainStreet;


    ///The direction of traffic flow on the main street.

    IENTCSData::Direction mainStreetDirection;


    ///Latitude of the controller in micro degrees.  Positive if north of

    ///equator, negative if sourth.

    long     latitude;


    ///Longitude of the controller in micro degrees.  Positive if east of

    ///prime meridian, negative if west.

    long     longitude;
```

```
        ///Name describing the type of controller.
        string   controllerType;

        ///Brief description of where this controller is. Usually a main street
        ///@ cross street specification.
        string   description;
    };
```

The controllerType member of the structure should be a human-understandable name of the controller and firmware in the controller. Some example controller type strings are listed in Table 3-3 below.

Table 3-3: Example Controller Type Strings

EXAMPLE CONTROLLER TYPE STRINGS
BI Tran 207LRT
BI Tran 222WP
BI Tran CIC
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
LACO-4E

The accompanying *IntersectionInfoSeq* sequence defines a list of *IntersectionInfo* structures.

```
typedef sequence<IntersectionInfo> IntersectionInfoSeq;
```

3.1.7.2.2 IntersectionRTSummary Structure

This structure contains slowly changing state information for an intersection controller. It is denoted by the IEN_INTERSECTIONRTSUMMARY enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_ INTERSECTIONRTSUMMARY event defined in Version 2 of the interface.

```
struct IntersectionRTSummary

{

    ///Id of the controller.

    MCSDATA::MCSDeviceID                    id;


    ///Timing plan selection mode in use by the controller.

    IENTCSData::SignalControlMode           controlMode;


    ///Operational state of the controller.

    IENTCSData::IntersectionSignalState     signalState;


    ///Controller response state, RESPONDING if controller responded to last

    ///communication attempt, NOT_RESPONDING if not, UNKNOWN if system

    ///problem prevents communication.

    IENTCSData::ControllerResponseState     responseState;


    ///If preemption in effect, cause for preemption. NO_PREEMPT if

    ///preemption not in effect.

    IENTCSData::PreemptType                 preemptType;


    ///A bitmask representing one or more active alarms for the controller.

    long                                    alarms;


    ///True if main street green is active, false if inactive or unknown.

    boolean                                 isMainStreetGreen;
```

```
        ///Overall state of communication from central to the controller,
        ///COMM_GOOD if comm mostly functioning as desired, COMM_BAD if quality
        ///of comm is too low for use by central, UNKNOWN if central problem
        ///prevents communication.
        IENTCSData::CommState                   commState;

        ///Numeric id of the timing plan.
        long                                    timingPlanID;

        ///Desired cycle length in seconds.
        long                                    desiredCycleLength;

        ///Desired offset in seconds.
        long                                    desiredOffset;

        ///Observed offset in seconds, -1 if unknown.
        long                                    actualOffset;
    };
```

The *IntersectionRTSummarySeq* sequence defines a list of *IntersectionRTSummary* structures.

```
typedef sequence<IntersectionRTSummary> IntersectionRTSummarySeq;
```

3.1.7.2.3 IntersectionRTStatus Structure

This structure contains rapidly changing state information for an intersection controller. It is denoted by the IEN_INTERSECTIONRTSTATUS enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_ INTERSECTIONRTSTATUS event defined in Version 2 of the interface.

```
    struct IntersectionRTStatus
    {
        //ID of the controller.
        MCSDATA::MCSDeviceID                    id;

        //Seconds since the start of current cycle.
```

```
        short                                   cycleCounter;


        //Cycle counter for controller with the same cycle length but offset 0
        short                                   referenceCycleCounter;
    };
```

The accompanying *IntersectionRTStatusSeq* sequence defines a list of *IntersectionRTStatus* structures.

```
    typedef sequence<IntersectionRTStatus> IntersectionRTStatusSeq;
```

3.1.7.2.4 PhaseStateData Structure

This structure contains the numeric identifiers of phases that are currently active in an intersection controller. It is denoted by the IEN_PHASE_STATEDATA enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_PHASE_STATEDATA event defined in Version 2 of the interface.

```
    struct PhaseStateData
    {
        //ID of the controller.
        MCSDATA::MCSDeviceID id;


        //Numeric IDs of phases that are currently green.  The sequence length
        //should be equal to the number of phases that are currently active. If
        //there are none then this sequence should contain one element with a
        //value of zero.
        IEN::ShortSeq activeGreens;
    };
```

The accompanying *PhaseStateDataSeq* sequence defines a list of *PhaseStateData* structures.

```
    typedef sequence<PhaseStateData> PhaseStateDataSeq;
```

3.1.7.2.5 PedestrianPhaseState Structure

This structure contains the numeric identifiers of phases that are currently displaying walk signals in an intersection controller. It is denoted by the IEN_PEDPHASE_STATEDATA enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_PEDPHASE_STATEDATA event defined in Version 2 of the interface.

```
    struct PedestrianPhaseState
    {
```

```
        //Id of the controller
        MCSDATA::MCSDeviceID id;

        //Numeric IDs of phases that are currently displaying a walk signal.
        //The sequence length should be equal to the number of phases in walk.
        //If there are none then this sequence should contain one element with a
        //value of zero.
        IEN::ShortSeq activePeds;
    };
```

The accompanying *PedestrianPhaseStateSeq* sequence defines a list of *PedestrianPhaseState* structures.

```
    typedef sequence<PedestrianPhaseState> PedestrianPhaseStateSeq;
```

3.1.7.2.6 VehicleCallState Structure

This structure contains the numeric identifiers of all phases that have vehicle calls active in an intersection controller. It is denoted by the IEN_VEHCALL_STATEDATA enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_VEHCALL_STATEDATA event defined in Version 2 of the interface.

```
    struct VehicleCallState
    {
        //ID of the controller.
        MCSDATA::MCSDeviceID id;

        //Numeric IDs of phases with active vehicle calls.  The length of the
        //sequence should be the number of phases that have active vehicle
        //calls.  If there are none then the length should be one with a single
        //phase value of zero.
        IEN::ByteSeq phasesWithCalls;
    };
```

The accompanying *VechicleCallStateSeq* sequence defines a list of *VehicleCallState* structures.

```
    typedef sequence<VehicleCallState> VehicleCallStateSeq;
```

3.1.7.2.7 PhaseTime Structure

The *PhaseTime* structure is used to report green times in seconds for a controller, both planned and observed. The accompanying *PhaseTimeSeq* sequence defines a list of *PhaseTime* structures, and is used in both the *LastCyclePhaseData* and *TpPhaseData* structures. This structure was added for Version 3 of the CDI definition.

```
struct PhaseTime

{

    //Numeric identifier for a phase.

    short phaseId;


    //Number of seconds that the phase was or should be active.

    short phaseTime;

};


typedef sequence<PhaseTime> PhaseTimeSeq;
```

3.1.7.2.8 LastCyclePhaseData Structure

This structure contains the numeric identifiers and green time in seconds of all phases that were active in the controller's most recently completed cycle. It is denoted by the IEN_LASTCYCLE_PHASEDATA enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_LASTCYCLE_PHASEDATA event defined in Version 2 of the interface.

Note that IEN user interface programs require that the CDI always return the same number of elements in the greenTimes sequence, regardless of how many phases were active in the controller's last cycle. As a guide, the CDI should return times for the maximum number of phases that the controller supports. The CDI should return a phase time of 0 for inactive phases.

```
struct LastCyclePhaseData

{

    //ID of the controller.

    MCSDATA::MCSDeviceID     id;


    //The total time of all the active phases;

    long         totalPhaseTime;
```

```
        //Green times for phases that were active in the controller's last
        //completed cycle.  The length of the sequence should be the number of
        //phases that were active in the last cycle. Each element contains the
        //ID of an active phase and the total time in seconds that the phase was
        //active in the last completed cycle. If the totalPhaseTime field is
        // zero then the length of this sequence should be zero as well.
        PhaseTimeSeq greenTimes;
    };
```

The accompanying *LastCyclePhaseDataSeq* sequence defines a list of *LastCyclePhaseData* structures.

```
    typedef sequence<LastCyclePhaseData> LastCyclePhaseDataSeq;
```

3.1.7.2.9 TpPhaseData Structure

This structure contains phase IDs and planned phase times in seconds for a controller's current timing plan. It is denoted by the IEN_TP_PHASEDATA enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_TP_PHASEDATA event defined in Version 2 of the interface.

```
    struct TpPhaseData
    {
        //Id of the controller.
        MCSDATA::MCSDeviceID id;

        //Planned phase times in seconds for the current timing plan in the
        //controller.  The length of the sequence should be the number of phases
        //active in the timing plan.  Each element in the sequence should
        //contain the phase ID and the planned phase time in seconds.  The
        //sequence length may be zero if the controller is currently running
        //free or is in flash.
        PhaseTimeSeq         plannedPhaseTimes;
    };
```

The accompanying *TpPhaseDataSeq* sequence defines a list of *TpPhaseData* structures.

```
    typedef sequence<TpPhaseData> TpPhaseDataSeq;
```

3.1.7.2.10 DetectorInfo Structure

This structure contains configuration information for a system detector. It is denoted by the IEN_DETECTORINFO enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_DETECTORINFO event defined in Version 2 of the interface.

```
struct DetectorInfo
{
    ///Id of the detector.
    MCSDATA::MCSDeviceID       id;

    ///Averaging period in seconds used to create averaged detector data. If
    ///averaged data is not available, should be -1.
    long                       averagingPeriod;

    ///Usage of the detector in the traffic system.
    IENTCSData::DetectorClass  detectorClass;

    /// Vehicle detection technique used by the detector.
    IENTCSData::DetectorType   detectorType;

    /// Direction of traffic flow past the detector.
    IENTCSData::Direction      detectorDirection;

    //Lane number for traffic passing the detector. The innermost lane
    //on the roadway is lane 1, the next lane to the right is lane 2, etc.
    octet                      laneNumber;

    //Name of the roadway that contains the detector.
    string                     roadName;

    //Weighting factor(k) for volume+weighted occupancy calculations.
    float                      weightingFactor;
```

```
};
```

The accompanying DetectorInfoSeq defines a list of DetectorInfo structures.

```
typedef sequence<DetectorInfo> DetectorInfoSeq;
```

3.1.7.2.11 DetectorState Structure

This structure contains the latest state information known by the TCS for a system detector, along with the time at which the detector information was gathered from the field device (as opposed to the time at which the IEN Site Server requested the data from the CDI or the CDI collected the data from the TCS). It is denoted by the IEN_DETECTORSTATE enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_DETECTORSTATE event defined in Version 2 of the interface.

```
struct DetectorState

{

    ///Id of the detector.

    MCSDATA::MCSDeviceID       id;


    ///The time of the updload in HHMMSS format: hours (0-23 for midnight to

    ///11 p.m.) * 10,000 | minutes (0-59) * 100 | seconds (0-59)

    IEN::HHMMSS                lastUpdateTime;


    ///The date of the update in YYYYMMDD format: year (AD) * 10,0000 |

    ///month (1-12) | day of month (1-31).

    IEN::YYYYMMDD              lastUpdateDate;


    ///Volume reported by the detector in vehicles per hour.

    long                       volume;


    /// Average volume (-1 if not available).

    long                       avgVolume;


    ///Volume in veh. per hour + weighted occupancy (-1 if not available).

    long                       volumePlusWeightingFactor;
```

```
        ///Avg volume + weighted occupancy. (-1 if not available).
        long                        avgVolumePlusWeightingFactor;

        ///Latest known status of the detector.  If the field is a value other
        ///than DETECTOR_OPERATIONAL then the volume, occupancy, and speed
        ///fields should be set to -1.
        IENTCSData::DetectorStatus status;

        ///Speed in miles per hour.  -1 if not available.
        short                       speed;

        ///Average speed over averaging period.  -1 if not available.
        short                       avgSpeed;

        ///Occupancy percentage.
        short                       occupancy;

        ///Average occupancy.  -1 if not available.
        short                       avgOccupancy;
    };
```

The accompanying *DetectorStateSeq* sequence defines a list of *DetectorState* structures.

```
typedef sequence<DetectorState> DetectorStateSeq;
```

3.1.7.2.12 SectionInfo Structure

This structure contains configuration information for a section. It is denoted by the IEN_SECTIONINFO enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_SECTIONINFO event defined in Version 2 of the interface.

```
struct SectionInfo

{

    //ID of section.
```

```
        MCSDATA::MCSDeviceID id;


        //The IDs of controllers that belong to this section.
        IEN::LongSeq         intersections;
    };
```

The accompanying *SectionInfoSeq* sequence defines a list of *SectionInfo* structures.

```
    typedef sequence<SectionInfo> SectionInfoSeq;
```

3.1.7.2.13 SectionState Structure

This structure contains state information for a section. It is denoted by the IEN_SECTIONSTATE enumeration value in the *IEN_EventType* enumeration. It replaces the IEN_SECTIONSTATE event defined in Version 2 of the interface.

```
    struct SectionState
    {
        //Id of the Section.
        MCSDATA::MCSDeviceID            id;


        ///Control mode used to select the timing plans of controllers following
        ///the section.
        IENTCSData::SignalControlMode   sectionControlMode;


        //Timing plan of the controllers that follow this Section.
        short                           planID;
    };
```

The *SectionStateSeq* sequence defines a list of *SectionState* structures.

```
    typedef sequence<SectionState> SectionStateSeq;
```

3.1.7.2.14 DetectorStationInfo and DetectorStationState Structures

The *DetectorStationInfo* structure, the *DetectorStationInfoSeq* typedef, the *DetectorStationState* structure, and the *DetectorStationStateSeq* typedef will be used for distribution of freeway data in a later version of

the IEN. The CDI should never receive requests for these data types from the Site Server, and should not return data of these types to the Site Server.

3.1.7.3 SiteUpdate Component Structures

The structures defined in this section are components of the *SiteUpdate* structure.

3.1.7.3.1 DeviceConfigGroup Structure

This component of the *SiteUpdate* structure contains configuration information for intersections, detectors, and sections, and an update time specified in hours, minutes, and seconds.

```
struct DeviceConfigGroup

{

    /// Time update was created by the CDI.

    IEN::HHMMSS         timeOfDistribution;


    /// Configuration information for intersections.

    IntersectionInfoSeq intersections;


    /// Configuration information for detectors.

    DetectorInfoSeq     detectors;


    /// Configuration information for sections.

    SectionInfoSeq      sections;


    /// Configuration information for detector stations

    DetectorStationInfoSeq stations;

};
```

The accompanying *DeviceConfigBatch* sequence, which is a sequence of *DeviceConfigGroup* structures, is one of the components of the *SiteUpdate* structure that the CDI should return from a call to its getDeviceUpdate32 method.

```
typedef sequence<DeviceConfigGroup>  DeviceConfigBatch;
```

3.1.7.3.2 DeviceStateGroup Structure

This structure contains slowly changing device state information for intersections, detectors, and sections, and an update time specified in hours, minutes, and seconds.

```
struct DeviceStateGroup
```

```
{
    /// Time update was created by the CDI.
    IEN::HHMMSS             timeOfDistribution;

    /// Intersection summary information (suitable for map).
    IntersectionRTSummarySeq intersectionStates;

    /// Detector state information (last set of uploaded data).
    DetectorStateSeq         detectorStates;

    /// Section state information (control mode and timing plan in use).
    SectionStateSeq          sectionStates;

    /// Freeway Detector Station states
    DetectorStationStateSeq stationStates;
};
```

The accompanying *DeviceStateBatch* sequence contains a sequence of *DeviceStateGroup* structures. It is one of the components of the *SiteUpdate* structure that the CDI should return from a call to its getDeviceUpdate32 method.

```
typedef sequence<DeviceConfigGroup> DeviceConfigBatch;
```

3.1.7.3.3 RealTimeUpdateGroup Structure

This structure contains rapidly changing intersection state information and an update time specified in hours, minutes, and seconds. The IEN Site Server requests the data types in this structure only if an IEN user interface program has subscribed to them and the Site Server has not exceeded the maximum configured number of subscriptions for the CDI.

```
struct RealTimeUpdateGroup
{
    /// Time update was created by the CDI.
    IEN::HHMMSS                 timeOfDistribution;

    /// Intersection cycle counters and master cycle counters.
```

```
        IntersectionRTStatusSeq       intersectionRTStatuses;

        /// Active green phases for intersections.
        PhaseStateDataSeq             intersectionPhaseStates;

        /// Active pedestrian walk signals for intersections.
        PedestrianPhaseStateSeq       pedestrianPhaseStates;

        /// Active vehicle calls for intersections.
        VehicleCallStateSeq           vehicleCallStates;

        /// Phase times from completed cycles for intersections.
        LastCyclePhaseDataSeq         lastCycles;

        /// Planned phase times for intersections.
        TpPhaseDataSeq                maximumPhaseTimes;
    };
```

The accompanying *RealTimeUpdateBatch* sequence contains a sequence of *RealTimeUpdateGroup* structures. It is one of the components of the *SiteUpdate* structure that the CDI should return from a call to its getDeviceUpdate32 method.

3.1.7.4 SiteUpdate Structure

The *SiteUpdate* structure is the structure in which the CDI should return requested device data when the IEN Site Server calls the getDeviceUpdate32 method of the CDI's *MCSDataAccessor* interface. The CDI is responsible for setting the size and filling in the configUpdates, stateUpdates, and realTimeUpdates sequences in the *SiteUpdate* structure. It is not responsible for filling in the site, system, or size_of_data fields in the structure.

```
    struct SiteUpdate
    {
        /// Display ID of site of origin for the update.  Filled in by the
        /// site server, not the CDI.
        short              site;
```

```
        /// Display ID of system of origin for the update.   Filled in by the
        /// site server, not the CDI.
        short               system;

        /// Size of the data contained in the update in bytes.  Filled in by the
        /// site server after getting the update from the CDI.
        long                size_of_data;

        /// Requested configuration information for devices on the TCS.
        DeviceConfigGroup   configUpdates;

        /// Requested device state information for the TCS.
        DeviceStateGroup    stateUpdates;

        /// Requested intersection real-time state information for the TCS.
        RealTimeUpdateGroup realTimeUpdates;
    };
```

3.1.7.5 Structures and Sequences Used in MCSDataAccessor Methods

This section defines other structures and sequences used in *MCSDataAccessor* methods.

3.1.7.5.1 MCSDeviceDataTypes Structure

The *MCSDeviceDataType* structure defines the set of event types that a CDI supports for a given device type.

```
    struct MCSDeviceDataTypes
    {
        /// Type of device that the CDI supports
        IENRTData::DeviceType  type;

        /// Types of events that a site server may request for the device type.
```

```
        MCSDATA::EventTypeList eventTypes;
    };
```

3.1.7.5.2 MCSDeviceDataTypeList Sequence

The *MCSDeviceDataTypeList* sequence, a sequence of *MCSDeviceDataType* structures, is returned by the CDI's getDeviceEventTypes method. The Site Server calls this entry point to request the set of device and event types supported by the CDI.

```
typedef sequence<MCSDeviceDataTypes> MCSDeviceDataTypeList;
```

3.1.7.5.3 MCSDeviceRequest Structure

The *MCSDeviceRequest* structure defines a request for one or more types of data for a single device.

```
struct MCSDeviceRequest
{
    /// Device type and 32 bit device ID.
    MCSDATA::MCSDevice      device;

    /// List of event types to request for the device.
    MCSDATA::EventTypeList  eventTypes;

    /// True if requesting changes since the last time the device's data
    /// was requested, false if the CDI should return all data for the
    /// device regardless of what data was last reported by the CDI.
    boolean                 changedOnly;
};
```

3.1.7.5.4 MCSDeviceRequestList Sequence

The *MCSDeviceRequestList* sequence contains one or more *MCSDeviceRequest* structures. The IEN Site Server passes an instance of this sequence to each call of the CDI's getDeviceEventData32 method to indicate the devices and data types currently of interest to the IEN.

```
typedef sequence<MCSDeviceRequest> MCSDeviceRequestList;
```

3.1.7.5.5 ReasonForRequestFailure Enumeration

The CDI uses this enumeration to report problems responding to a data request from the IEN Site Server.

```
enum ReasonForRequestFailure
{
```

```
        /// requested event type not supported
        EVENT_TYPE_NOT_SUPPORTED,

        /// access to data for device denied
        ACCESS_DENIED,

        /// requested device does not exist
        NO_DEVICE,

        /// other unspecified failure
        OTHER
    };
```

3.1.7.5.6 RequestFailureStatus Structure

The CDI uses this structure to report the failure of one kind of event for one device.

```
    struct RequestFailureStatus
    {
        // Device for which request failed.
        MCSDATA::MCSDevice          device;

        // Type of data requested.
        IENTCSData::IEN_EventType   eventType;

        // Reason data request failed.
        ReasonForRequestFailure     reason;
    };
```

3.1.7.5.7 RequestFailureStatusSeq Sequence

This is a sequence of *RequestFailureStatus* structures. The CDI returns a list of these sequences from its getDeviceEventData32 method if any of the device/data type pairs requested by the IEN Site Server could not be returned for one of the following reasons:

- The CDI does not support the requested data type for the given device type;
- The CDI cannot access the requested device due to security or other restrictions;

- The device does not exist;
- Some other internal problem.

3.1.7.6 MCSDataAccessor Interface

The *MCSDataAccessor* interface contains replacements for the deviceDataTypes, getDeviceList, and getDeviceEventDataList methods that use 32-bit device identifiers and enumerated values for requested data types.

```
interface MCSDataAccessor : TCSData::DataAccessor {

    /// Get the event types supported for all device types

    /// for which this CDI returns data.

    /// @return       supported data types for all supported devices

    MCSDeviceDataTypeList getDeviceEventTypes( );


    ///Get a list of devices accessible to the IEN of the given types.

    ///@return The 32 bit version of the devices currently configured to

    ///report to the IEN.

    MCSDATA::MCSDeviceList getAvailableDevices32(

        in TCS::DeviceTypeList types);


    /// Request TCS data by device and event type from the CDI.  The CDI

    /// should return all the data that it can in the set_update structure.

    /// For those requests that the CDI cannot satisfy, it should report the

    /// device ID, type and reason the request failed in the failed_requests

    /// sequence.

    ///

    /// @param device_requests    List of devices and data types requested

    /// @param site_update        Requested TCS data for those devices and

    ///                           event types that the CDI can satisfy.  The

    ///                           site server fills in the site, system, and

    ///                           size_of_data fields.  The CDI fills in the

    ///                           configUpdates, stateUpdates, and

    ///                           realTimeUpdates sequences.  If there is no
```

```
        ///                             data to report, the CDI may set the length
        ///                             of these sequences to zero.
        ///                             No data should be reported for failed
        ///                             requests or for change-only requests for
        ///                             which the device's configuration or state
        ///                             hasn't changed since the last report.
        ///                             The CDI should report the reasons for
        ///                             failed requests in the failed_requests
        ///                             parameter.
        /// @param failed_requests      Reasons for failed requests. Should have
        ///                             one element for each request in
        ///                             device_requests for which there is an
        ///                             exceptional failure.  No data reported to
        ///                             change-only requests is not considered a
        ///                             failure and should not be reported here.
        void getDeviceUpdate32(in MCSDeviceRequestList device_requests,
                               inout SiteUpdate site_update,
                               out RequestFailureStatusSeq failed_requests)
                                 raises (TCS::SystemStatusException,TCS::Error);
    };
```

The IEN Site Server will periodically call the `getAvailableDevices32` method of the *MCSDataAccessor* object to get the set of devices that it should request from the CDI. It will call the `getDeviceUpdate32` method once per second to get data on the available devices. Most requests that it makes will have the changedOnly flag set, to reduce the traffic sent over the IEN. Additionally, the IEN Site Server will not request rapidly changing intersection data, which is to say any of the device data contained in the *RealTimeUpdateGroup* structure, from the CDI unless a user interface application is requesting a real-time view of an intersection. The applications that can show real-time intersection views are ATMS Explorer diagrams and the IEN Intersection Monitor window.

3.1.8 TCSCommand.idl

The `TCSCommand.idl` file defines the TCSCommand module. It depends on the `TCS.`idl and `IENRtData.idl` files. It defines the command interface for a version 2 CDI. This section of the document lists the items in the `TCSCommand.idl` file that are still used for version 3 of the CDI.

3.1.8.1 CommandsNotAccepted Exception

The command interface throws this exception to indicate that it is not accepting commands from the IEN. The `reason` field is a free-form string that the CDI may set to report further details about why commands are not accepted.

```
exception CommandsNotAccepted

{

    string reason;

};
```

3.1.8.2 CommandAccessor Interface

The `setCDIPlan`, `changeMode`, and `releaseControl` methods in this interface have been replaced in the new *MCSCommandAccessor* interface. However, the CDI should still implement the `clientName` attribute and the `destroy` method.

```
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
        );


    };
```

3.1.8.3 CommandAccessorFactory Interface

The CDI must implement an object with this interface. The IEN Site Server must call the CDI's *CommandAccessorFactory* to get an instance of a *CommandAccessor*. A Version 3 CDI should implement a *CommandAccessorFactory* that returns its *MCSCommandAccessor* cast to the old *CommandAccessor* interface. The Site Server will check the version information for the *CommandAccessor*; if the version information has a major revision of 3 or higher, it will try to cast the *CommandAccessor* to an instance of *MCSCommandAccessor*.

```
    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);

    };
```

3.1.9 MCSCommand.idl

The `MCSCommand.idl` file defines the MCSCommand module. It depends on the `TCSCommand.idl`, `TCS.idl` and `IENRtData.idl` files. It defines the *MCSCommandAccessor* interface, which the IEN Site Server program calls to send commands from the IEN to traffic control devices on a TCS, and several exceptions and structures used in methods that send commands to the TCS.

3.1.9.1 MCS Command Interface Version Numbers

The constants defined in the beginning of the MCSCommand module are the major version number, minor version number, and revision number that should be returned by an MCSCommandAccessor object to be recognized as an instance of the MCSCommandAccessor object by IEN Site Server programs.

```
    const short majorVersion = 3;

    const short minorVersion = 0;

    const short revision     = 0;
```

3.1.9.2 CommandResult Structure

The *CommandResult* structure contains the result for executing a command on a single device. The accompanying *CommandResultSequence* is used in the *MCSCommandAccessor* interface to report the results of commands.

```
    struct CommandResult {

        /// Type and ID of commanded device.

        MCSDATA::MCSDevice device;


        /// True if command successful, false if not.

        boolean isSuccessful;


        /// Free-form string for reporting the reason the command failed.

        string message;

    };


    typedef sequence<CommandResult> CommandResultSequence;
```

3.1.9.3 Command Exceptions

The following exceptions can be thrown by individual commands. They report 32-bit device IDs of the devices for which the commands failed.

```
    /// Thrown if devices are specified that
    /// are unknown to TCS or not fully configured
    exception MCSUnknownDevices
    {
        MCSDATA::MCSDeviceList unknowns;
    };

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

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

    /// Thrown when a command is not supported by the CDI for one or more of the
    /// requested devices.
    exception MCSCommandNotSupported
```

```
{
    MCSData::MCSDeviceList devices;
}
```

3.1.9.4 MCSCommandAccessor Interface

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

```
interface MCSCommandAccessor: TCSCommand::CommandAccessor
{
    ///Get a list of devices of the given types to which the IEN may give
    ///commands.
    ///
    ///@param types device types in which the caller is interested
    ///
    ///@return The 32 bit version of the devices that may currently accept
    ///        commands from the IEN
    MCSDATA::MCSDeviceList getAvailableDevices32(
        in TCS::DeviceTypeList types);


     /// Change CDI plan number using 32-bit device IDs.
    ///
    /// @param devices    List of devices for which to change the plan
    /// @param planNumber New plan number for the requested devices
    /// @return           Results for command to each device on list
    /// @throws           CommandsNotAccepted if
    CommandResultSequence setCDIPlan32(
        in MCSDATA::MCSDeviceList devices,
        in short                  planNumber
        ) raises (
            TCSCommand::CommandsNotAccepted,
```

```
            MCSInvalidPlanNumber,
            MCSUnknownDevices,
            MCSCommandNotSupported,
            TCS::SystemStatusException,
            TCS::Error
        );

    /// Change operational mode of specified devices using 32-bit device
    /// IDs.
    ///
    /// @param devices    List of devices for which to change the mode with
    ///                   32-bit device IDs.
    /// @param newMode    New operational mode for the devices on the list
    /// @return           Results for command to each device on list
    CommandResultSequence changeMode32(
        in MCSDATA::MCSDeviceList devices,
        in TCS::Mode              newMode
        ) raises (
            TCSCommand::CommandsNotAccepted,
            MCSInvalidMode,
            MCSUnknownDevices,
            MCSCommandNotSupported,
            TCS::SystemStatusException,
            TCS::Error
        );

    /// Release IEN control of a list of TCS devices using 32-bit device
    /// IDs.
    ///
    /// @param devices    List of devices for which to release IEN control
```

```
        ///                  with 32-bit device IDs
        /// @return          Results for command to each device on list
        CommandResultSequence releaseControl32(
            in MCSDATA::MCSDeviceList devices
            ) raises (
                MCSUnknownDevices,
                MCSCommandNotSupported,
                TCS::SystemStatusException,
                TCS::Error
            );


        /// Put MCS devices into flashing operation.  Remove from flash by
        /// calling releaseControl32.
        ///
        /// @param devices  List of devices to put into flash, with 32-bit
        ///                 device IDs.
        /// @return         Results for flash command to each device on list
        CommandResultSequence flash32(
            in MCSDATA::MCSDeviceList devices
            ) raises (
                TCSCommand::CommandsNotAccepted,
                MCSCommandNotSupported,
                MCSUnknownDevices,
                TCS::SystemStatusException,
                TCS::Error
            );
    };
```

3.1.9.4.1 setCDIPlan32 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 and report the success or failure of commanding all devices in the list in the *CommandResultSequence* that the method returns. If it cannot implement the command on one of the requested devices due to an invalid plan number or an invalid device ID, it should raise an *MCSInvalidPlanNumber* or *MCSUnknownDevices* exception. If the CDI does not support this command, it should raise an *MCSCommandNotSupported* exception.

3.1.9.4.2 changeMode32 Method

changeMode32 changes the operating mode of the devices on the list to the requested mode. See Table 3-1 for a list of operating modes usable in change mode command to a section or intersection. (Note that the set of control modes available for commands differs from those that can be reported by a CDI.)

The CDI should make a "best effort" attempt to change the mode of the requested controllers and sections and report the success or failure of commanding all devices in the list in the *CommandResultSequence* that the method returns. If it cannot implement the command on one or more of the requested devices due to an invalid mode or unsupported control mode or an invalid device ID, it should raise an *MCSInvalidMode* or *MCSUnknownDevices* exception. If the interface does not support this command, it should raise an *MCSCommandNotSupported* exception.

3.1.9.4.3 releaseControl 32Method

This method ends external control of all devices on the requested list. It terminates plan changes, mode changes, or flash commands requested for the devices on the list. The CDI should return the devices to the state desired by the TCS when the CDI receives the releaseControl32 method call, and report the success or failure of releasing the requested controllers and sections from IEN control in the *CommandResultSequence* that the method returns. If the interface does not support this command, it should raise an *MCSCommandNotSupported* exception.

3.1.9.4.4 flash32 Method

This method starts flashing operation for a list of devices. The CDI should attempt to put the requested intersections into flash, and report the success or failure of the command for all devices in the list in the *CommandResultSequence* that the method returns. If it cannot implement the command on one of the requested devices due to an invalid device ID, it should raise an *MCSUnknownDevices* exception. If the interface does not support this command, it should raise an *MCSCommandNotSupported* exception.

3.2 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 passed from the IEN through the Site Server to the CDI.

3.2.1 Data Access Requirements

The TCS CDI must be able to completely process a call to the *MCSDataAccessor* interface's *getDeviceUpdate32* method within a half second.

The IEN Site Server requests data for most devices only if the data has changed since the last data request. The Site Server requests full device data, whether or not it has changed since the last request, on a different fraction (approximately 1/60th) of the TCS devices each second. Consequently, it requests full state and configuration data on all devices exported by the TCS once per minute.

3.2.2 Data Reporting Requirements

When the Site Server queries a CDI for device data, the CDI must return a *SiteUpdate* structure containing one or more device event structures with data for the requested devices, as described in Sections 3.1.7.1, 3.1.7.2.14 and 3.1.7.4. 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 an operator requests that they do so. 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 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 is capable of receiving data at whatever frequency the TCS supports.

Table 3-4 below summarizes the event types used by the IEN and their priority. Text below will discuss each event in detail.

Table 3-4: IEN Event Types

EVENT TYPE	ENTITY	WHEN REQUESTED	PRIORITY
IEN_COMMANDRETURN	Command	N/A	N/A
IEN_INTERSECTIONINFO	Intersection	Always	Required
IEN_INTERSECTIONRTSTATUS	Intersection	If Subscribed	Required
IEN_INTERSECTIONRTSUMMARY	Intersection	Always	Required Highly Desirable Desirable
IEN_PHASE_STATEDATA	Intersection	If Subscribed	Highly Desirable
IEN_PEDPHASE_STATEDATA	Intersection	If Subscribed	Desirable
IEN_VEHCALL_STATEDATA	Intersection	If Subscribed	Desirable

EVENT TYPE	ENTITY	WHEN REQUESTED	PRIORITY
IEN_LASTCYCLE_PHASEDATA	Intersection	If Subscribed	Highly Desirable
IEN_TP_PHASEDATA	Intersection	If Subscribed	Highly Desirable
IEN_DETECTORINFO	Detector	Always	Required (If Supported)
IEN_DETECTORSTATE	Detector	Always	Required (If Supported)
IEN_SECTIONINFO	Section	Always	Required (If Supported)
IEN_SECTIONSTATE	Section	Always	Required (If Supported)

3.2.2.1 IEN_INTERSECTIONINFO

The CDI is required to return an *IntersectionInfo* structure for an intersection controller when the Site Server requests an event of type IEN_INTERSECTIONINFO for an intersection.

The IEN Site Server requests this type of data once per second for all configured intersections, with the `changedOnly` flag set to false approximately once per minute.

3.2.2.2 IEN_INTERSECTIONRTSTATUS

It is highly desirable that the CDI return an *IntersectionRTStatus* structure for an intersection controller when the Site Server requests an event of type IEN_INTERSECTIONRTSTATUS for an intersection controller.

The IEN Site Server requests this type of data once per second for an intersection if one or more IEN user interface programs have subscribed to real-time intersection data for that intersection.

3.2.2.3 IEN_INTERSECTIONRTSUMMARY

The CDI is required to return a structure of type *IntersectionRTSummary* when the Site Server requests an event of type IEN_INTERSECTIONRTSUMMARY for an intersection. 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 of the communication status values indicates 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 NO_PREEMPT if the controller's preemption type is not known by the TCS at the time of the request.

The IEN Site Server requests this type of data once per second for all configured intersections, with the `changedOnly` flag set to false once per minute.

3.2.2.4 IEN_PHASE_STATEDATA

It is highly desirable for the CDI to return a structure of type *PhaseStateData* when the Site Server requests an event of type IEN_PHASE_STATEDATA for an intersection controller. 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).

The IEN Site Server requests this type of data once per second for an intersection if one or more IEN user interface programs have subscribed to real-time intersection data for that intersection.

3.2.2.5 IEN_PEDPHASE_STATEDATA

It is desirable for the CDI to return a structure of type *PedestrianPhaseState* when the Site Server requests an event of type IEN_PEDPHASE_STATEDATA for an intersection controller. 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.

The IEN Site Server requests this type of data once per second for an intersection if one or more IEN user interface programs have subscribed to real-time intersection data for that intersection.

3.2.2.6 IEN_VEHCALL_STATEDATA

It is desirable for the CDI to return a structure of type *VehicleCallState* when the Site Server requests an event of type IEN_VEHCALL_STATEDATA for an intersection controller. 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.

The IEN Site Server requests this type of data once per second for an intersection if one or more IEN user interface programs have subscribed to real-time intersection data for that intersection.

3.2.2.7 IEN_LASTCYCLE_PHASEDATA

It is highly desirable that the CDI should return a structure of type *LastCyclePhaseData* when an event of type IEN_LASTCYCLE_PHASEDATA is requested by the Site Server for an intersection controller. 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.

The IEN Site Server requests this type of data for an intersection once per second if one or more IEN user interface programs have subscribed to real-time intersection data for that intersection.

3.2.2.8 IEN_TP_PHASEDATA

It is highly desirable that the CDI should return a structure of type *TpPhaseData* when an event of type IEN_TP_PHASEDATA is requested by the Site Server for an intersection controller. The CDI should update this data when the controller changes the planned green times allotted to any of its phases, such as when it changes timing plans or phase timing parameters.

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

The IEN Site Server requests this type of data for an intersection once per second if one or more IEN user interface programs have subscribed to real-time intersection data for that intersection.

3.2.2.9 IEN_DETECTORINFO

If the CDI supports any system detectors, it is required to return a structure of type *DetectorInfo* when the Site Server requests an event of type IEN_DETECTORINFO for a system detector. If a TCS does not natively support volume plus weighted occupancy data, it is highly desirable for the CDI to calculate V+kO and average V+kO values using a value of 30 for the weighting factor.

The IEN Site Server requests this type of data for all configured detectors once per second, with the `changedOnly` flag set to false approximately once per minute.

3.2.2.10 IEN_DETECTORSTATE

If the CDI supports any system detectors, it is required to return a structure of type *DetectorState* when the Site Server requests an event of type IEN_DETECTORSTATE for a system detector. The CDI should return new information in this event each time after the TCS uploads data from a system detector. As noted in the previous section, if a TCS does not natively support V+kO data, it is highly desirable for the CDI to calculate this data itself.

The IEN Site Server requests this type of data for all configured detectors once per second, with the `changedOnly` flag set to false approximately once per minute.

3.2.2.11 IEN_SECTIONINFO

If the CDI supports sections, it is required to return a structure of type *SectionInfo* when the Site Server requests an event of type IEN_SECTIONINFO for a section.

The IEN Site Server requests this type of data for all configured sections once per second, with the `changedOnly` flag set to false approximately once per minute.

3.2.2.12 IEN_SECTIONSTATE

If the CDI supports sections, it is required to return a structure of type *SectionState* when the Site Server requests an event of type IEN_SECTIONSTATE for a section.

The IEN Site Server requests this type of data for all configured sections once per second, with the `changedOnly` flag set to false approximately once per minute.

3.2.3 Command Execution Requirements

The TCS CDI is required to process the following method calls to its *MCSCommandAccessor* interface within 10 seconds:

- setCDIPlan32
- changeMode32
- releaseControl32
- flash32

3.3 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.

3.3.1 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.

3.3.2 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 3-5.

Table 3-5: CDI Names in the Naming Service

OBJECT	ID	KIND
CommandAccessorFactory	TCSCDICmd<Site_ID>:<System_ID>	Site<Site_ID>
DataAccessorFactory	TCSCDIData<Site_ID>:<System_ID>	Site<Site_ID>

The <Site_ID> string should be replaced with the ID number assigned to the IEN site at which the TCS CDI is running by Los Angeles County Department of Public Works (LACDPW). The <System_ID> string should be replaced with the ID number assigned by LACDPW to the particular TCS system at the site.

3.4 FIREWALL CONSIDERATIONS

In case a CDI must be installed at a site at which there is a firewall between the IEN Site Server and the computer on which the CDI runs, it should be possible to configure a fixed IP port on which the CDI creates its CORBA objects. This enables configuration of the firewall to allow communication between the IEN Site Server and the CDI on the fixed port.

Most ORBs support configuring a process to register CORBA objects on a fixed port, but the exact configuration method varies with different ORB implementations. For example, JacORB uses a configuration file named jacorb.properties, OmniORB uses a configuration file named omniorb.cfg, and TAO uses command line options and a configuration file named svc.conf. Consult the documentation of each ORB for exact location, name, and contents of the relevant configuration commands.

4. EXAMPLE IMPLEMENTATION

This section presents a sample CDI implementation based on the *TransSuite*® TCS CDI. TCS CDI implementations for other traffic control systems may vary, but they must comply with the TCS CDI interface requirements that were described above.

4.1 CORBA ORB USED

The IEN Site Server process runs on Microsoft Windows on an Intel Pentium-compatible processor. It uses version 2.3 of the JacORB ORB, which is a Java ORB freely available at http://www.jacorb.org/index.html.

The TransSuite® TCS CDI runs on Windows XP, Server 2003, or later system on an Intel Pentium-compatible system. It also uses version 2.3 of JacORB.

4.2 IMPLEMENTATION ENVIRONMENT

TransCore wrote the *TransSuite*® TCS CDI in the Java programming language. It runs as multiple threads in the *TransSuite*® TCS server process. A main CDI thread reports TCS device data changes to the CDI and handles publishing of names of the MCSDataAccessor and MCSCommandAccessor objects to the naming service.

4.3 CONFIGURATION DATA

The *TransSuite*® TCS CDI gets configuration data from shared objects that it receives from the main TCS process. The configuration data contains the site server node name and site ID to identify the TCS CDI on the network, and the data for intersection controllers, system detectors, and sections that the CDI will export to the IEN.

4.4 IMPORTANT CDI METHODS

All of the code shown in this section comes from the main CDI class in the *TransSuite*® TCS CDI. The class is shown in its entirety in the section named Appendix B: *Example* CDI Main Class below.

4.4.1 CDI Constructor

The constructor creates the data accessor factory and command accessor factory objects that will be published in the CDI's main thread.

```
public IEN_CDI() {

    tcs_command_factory_ = new TCS_CommandAccessorFactory(getPoa());

    tcs_data_factory_ = new TCS_DataAccessorFactory(getPoa());

}
```

4.4.2 TransSuite® TCS Main CDI Thread

This section describes the main thread for the *TransSuite*® TCS CDI.

```
public void run() {
```

```
System.out.printf("%s %s","ien", "IEN Component has started running.");

NamingContextExt naming_service = null;

try {
```

In the section below, the CDI creates the command and data accessor factory objects.

```
data_factory_ = getPoa()

        .servant_to_reference(tcs_data_factory_);

command_factory_ = getPoa().servant_to_reference(

        tcs_command_factory_);
```

The routine then enters the main TCS loop, in which it locates the naming service and republishes the names of the command and data factories, and then calls the performWork method to accept CORBA calls from the site server.

```
While (!Thread.currentThread().isInterrupted()) {

    naming_service = resolveNamingContext();

    System.out.printf("%s %s","ien", "IEN NamingContext resolved.");

    if (bindFactories(naming_service)) {

        System.out.printf("%s %s", "Informational",

                        "All factories published. Waiting for IEN I/O.");

        System.out.printf("%s %s", "ien",

                        "IEN Factories bound to the naming service.");


        performWork(naming_service);
```

After the CDI determines that it must republish names, it deactivates any active accessor objects to prepare for republication.

```
        tcs_data_factory_.deactivateAccessors();

        tcs_command_factory_.deactivateAccessors();

    } else {

        Thread.sleep(5000);

    }

}
```

Here the CDI catches the Java InterruptedException indicating that it should exit from the loop, along with two CORBA exceptions, ServantNotActive and WrongPolicy.

```
        } catch (InterruptedException e) {

            // Just exit from the loop

        } catch (ServantNotActive e) {

            e.printStackTrace();

        } catch (WrongPolicy e) {

            e.printStackTrace();

        }
```

Finally, the CDI removes the accessor factory objects from the name service:

```
        cleanup(naming_service);


        System.out.printf("%s %s","ien", "IEN Component has stopped.");

    }
```

4.4.3 resolveNamingContext method

The CDI uses this method to locate the CORBA naming service on the IEN Site Server. It waits until the thread is interrupted or it successfully locates the naming service.

```
    private NamingContextExt resolveNamingContext() throws InterruptedException {

        System.out.printf("%s %s","ien",

                "Resolving the IEN Site servers NameService");

        NamingContextExt naming_service = null;

        String error = null;

        int total_tries = 0;

        ORB orb = getOrb();

        while (!Thread.currentThread().isInterrupted()

                && naming_service == null) {

            String name_spec = "corbaloc:iiop:"

                    + siteServerName + ":14444/NameService";

            total_tries += 1;

            try {
```

```
            System.out.printf("%s %s %s",
                    "ien",
                    "Resolving nameserver for the site server at ",
                    siteServerName);
            org.omg.CORBA.Object obj = orb.string_to_object(name_spec);
            if (obj != null) {
                System.out.printf("%s %s",
                        "ien",
                        "ORB has resolved the initial reference "
                                + name_spec + " attempting narrow");
                naming_service = NamingContextExtHelper.narrow(obj);
            } else {
                error = "Unable to get initial reference to Name Service using "
                        + name_spec;
            }
        } catch (SystemException e) {
            error = "Unable to resolve naming service: "
                    + e.getLocalizedMessage();
        } catch (Exception e) {
            error = "Unable to resolve naming service: "
                    + e.getLocalizedMessage();
        }


        if (naming_service == null) {
            if (total_tries == 1 || total_tries % 600 == 0) {
                System.out.printf("%s %s", "Error",
                                "Unable to intialize CORBA Components.");
            }
            if (error != null) {
                System.out.printf("%s %s","ien", error);
```

```
                } else {
                    System.out.printf("%s %s",
                            "ien",
                            "Unable to determine why the name service could not be "
                                    + "resolved.");
                }
                error = null;
                Thread.sleep(10000);
            }
        }
        if (naming_service != null) {
            System.out.printf("%s %s","ien", "Obtained naming service");
        } else {
            throw new InterruptedException();
        }
        return (naming_service);
    }
```

4.4.4 bindFactories Method

This method binds the CDI's command accessor factory and data accessor factory objects to the proper names in the naming service. It catches all CORBA exceptions, in which case it will return "false" to the calling routine to indicate a problem interacting with the name service.

```
    private boolean bindFactories(NamingContextExt naming_service) {
        String error = null;
        boolean bound = false;
        try {
            String just_site_id = site_and_system_id_ ;
            int sys_ndx = just_site_id.indexOf(":");
            if (sys_ndx != -1) {
                just_site_id = just_site_id.substring(0, sys_ndx);
            }
```

```
            naming_service.rebind(new NameComponent[] { new NameComponent(
                    "TCSCDIData" + site_and_system_id_, "Site"
                            + just_site_id) }, data_factory_);
            naming_service.rebind(new NameComponent[] { new NameComponent(
                    "TCSCDICmd" + site_and_system_id_, "Site"
                            + just_site_id) }, command_factory_);
            bound = true;
        } catch (Exception e) {
            error = "Unable to bind factories Exception: "
                    + e.getClass().getCanonicalName() + " message "
                    + e.getLocalizedMessage();
        }

        if (error != null) {
            System.out.printf("%s %s","ien", error);
        }
        return (bound);
    }
```

4.4.5 performWork Method

The performWork method is the inner loop of the CDI, in which it passes control to the ORB and gives up the CPU to other threads in the TCS. It leaves this loop either if the naming service reference becomes unusable, the thread is interrupted, or an exception is thrown.

```
    private void performWork(NamingContextExt naming_service)
            throws InterruptedException
    {
        ORB orb = getOrb();
        try {
            while (!Thread.currentThread().isInterrupted()
                    && !naming_service._non_existent()) {
                if (orb.work_pending()) {
```

```
                orb.perform_work();
            }
            Thread.sleep(IEN_Waiting_Period);
            total_loops_ += 1;
        }
    } catch (SystemException e) {
        System.out.printf("%s %s: %s", "Error",
                "Exception occured while running the IEN interface",
                e.getLocalizedMessage());
        e.printStackTrace();
    }
}
```

4.4.6 cleanup method

This method is called when the CDI is shutting down. It deactivates any active data or command accessor objects, then unbinds the command and data accessor factory objects from the name service.

```
private void cleanup(NamingContextExt naming_service)
{
    if (data_factory_ != null) {
        tcs_data_factory_.deactivateAccessors();
        try {
            getPoa().deactivate_object(
                    getPoa().reference_to_id(data_factory_));
        } catch (Exception e) {
        }
    }

    if (command_factory_ != null) {
        tcs_command_factory_.deactivateAccessors();
        try {
            getPoa().deactivate_object(
```

```
                    getPoa().reference_to_id(command_factory_));
            } catch (Exception e) {
            }
        }
        if (naming_service != null) {
            unbindFactories(naming_service);
        }
    }
```

4.4.7 unbindFactories method

The unbindFactories method is called from the cleanup method to remove the data and command accessor factory object's names from the CORBA naming service.

```
    /**
     * Unbinds the factories from the naming service.
     *
     * @param naming_service
     *            NamingContextExt The naming context to unbind from.
     *
     * @return boolean True indicates that the data and command interfaces have
     *         been unbound from the given naming context false indicates that
     *         there was some problem unbinding the components from the name
     *         service
     */
    private boolean unbindFactories(NamingContextExt naming_service) {
        String error = null;
        boolean unbound = false;
        try {
            naming_service.unbind(new NameComponent[] { new NameComponent(
                    "TCSCDIData" + site_and_system_id_, "Site"
                            + site_and_system_id_) });
            naming_service.unbind(new NameComponent[] { new NameComponent(
```

```
                    "TCSCDICmd" + site_and_system_id_, "Site"
                            + site_and_system_id_) });
            unbound = true;
        } catch (Exception e) {
            error = "Unable to unbind the factories from the naming service: "
                    + e.getClass().getCanonicalName() + " message "
                    + e.getLocalizedMessage();
        }

        if (error != null) {
            System.out.printf("%s %s","ien", error);
        }
        return (unbound);
    }
```

5. APPENDICES

5.1 APPENDIX A: TCS CDI CORBA IDL FILES

5.1.1 IEN.idl

```
//--------------------------------------------------------------------------
// Copyright 2001, 2009 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
//--------------------------------------------------------------------------
// $Id: IEN.idl,v 1.7 2009/04/23 21:13:47 mayoj Exp $
//
// Core CORBA type and interface definitions for IEN
//--------------------------------------------------------------------------

#ifndef IEN_IDL
#define IEN_IDL

//#include "sctypes.idl"  // Showcase data types
#pragma prefix "transcore.com"
module IEN
{
    const string IDLFileID = "$Id: IEN.idl,v 1.6 2009/02/10 23:24:06 mayoj Exp $";

    // Define abbreviations for long type names
    typedef octet               Byte;
    typedef unsigned short      UShort;
    typedef unsigned long       ULong;

    // Define sequence types for all basic types
    typedef sequence<boolean>   BooleanSeq;
    typedef sequence<Byte>      ByteSeq;
```

```
typedef sequence<short>     ShortSeq;
typedef sequence<UShort>    UShortSeq;
typedef sequence<long>      LongSeq;
typedef sequence<ULong>     ULongSeq;
typedef sequence<float>     FloatSeq;
typedef sequence<double>    DoubleSeq;
typedef sequence<string>    StringSeq;
typedef sequence<any>       AnySeq;

// Define types of IEN system entities
// REVIEW_KDJ: Change this to an 'octet' type
enum EntityType
{
    ENTITYTYPE_NOT_AVAILABLE,       // none/unknown
    ENTITYTYPE_INTERSECTION,        // INT Intersection Controller
    ENTITYTYPE_DETECTOR,            // DET VSO Detector
    ENTITYTYPE_SECTION,             // SEC Group of intersections
    ENTITYTYPE_TCS,                 // TCS Traffic Control System
    ENTITYTYPE_LINK,                // LNK Roadway Link
    ENTITYTYPE_HOST,                // HST Workstation/Server
    ENTITYTYPE_DMS,                 // DMS Dynamic Message Sign
    ENTITYTYPE_CAMERA,              // CAM
    ENTITYTYPE_USER,                // USR
    ENTITYTYPE_ALARM,               // ALM
    ENTITYTYPE_INCIDENT,            // INC
    ENTITYTYPE_PLANNED_EVENT,       // PLN
    ENTITYTYPE_SCENARIO_PLAN,       // SCN Scenario Plan
    ENTITYTYPE_SCENARIO_ACTIVATION, // ACT Activated Scenario
    ENTITYTYPE_RAMP,                // RMP Ramp Meter
    ENTITYTYPE_HAR,                 // HAR Highway Advisory Radio
```

```
    ENTITYTYPE_ESS,                 // ESS Environmental Sensor Station
    ENTITYTYPE_CORRIDOR,            // COR Corridor
    ENTITYTYPE_SITE,                // SIT site
    ENTITYTYPE_USER_GROUP,          // GRP user group
    ENTITYTYPE_RESOURCE_GROUP       // RGP resource group
};

// Identification number of a corridor
typedef short CorridorID;
const CorridorID LOCAL_CORRIDOR = 0;

// Identification number of a site within a corridor
typedef short SiteID;
const SiteID LOCAL_SITE = 0;

// Identification number of a system within a site
typedef octet SystemID;
const SystemID LOCAL_SYSTEM = 0;

// Identification number of an entity of a particular type within a system
typedef long EntityNumber;
const EntityNumber NULL_ENTITY = 0;

// Unique identification information for an entity
struct EntityID
{
    CorridorID    corridor;  // zero means "local corridor"
    SiteID        site;      // zero means "local site"
    SystemID      system;    // zero means "local system"
    octet         type;      // value from EntityType enumeration
```

```
        EntityNumber  number;    // identification number
    };
    typedef sequence<EntityID> EntityIDSeq;

    typedef string<32> Identifier;
    const short IdentifierLenMax = 32;  // from TMDD
    typedef sequence<Identifier> IdentifierSeq;

    typedef string<64> Description;
    const short DescriptionLenMax = 64;
    typedef sequence<Description> DescriptionSeq;

    // Time and date representations follow the conventions used in the TMDD.
    // They are integers of the forms YYYYMMDD and HHMMSS.
    //
    // For example, May 19, 2000 would be 20000519 and 2:00 PM would be 140000.
    //
    // All times are UTC.

    // REVIEW_KDJ: Replace this with something standard?

    typedef long YYYYMMDD;
    typedef long HHMMSS;

    struct Timestamp
    {
        YYYYMMDD date;
        HHMMSS   time;
    };
    typedef sequence<Timestamp> TimestampSeq;
```

```
// Thrown when an attempt is made to use a string that
// is longer than is allowed
exception StringTooLong
{
    string reason;
    string details;
    string badString; // the string that caused the error
};

// Thrown when an attempt is made to use an empty string for
// a data value that requires a non-empty string
exception EmptyStringNotAllowed
{
    string reason;
    string details;
};

// Thrown when a request is made for an entity that does
// not exist
exception NoSuchEntity
{
    string   reason;
    string   details;
    EntityID entity;
};

// Thrown when a request is made for an entity that does
// not support the Entity object interface
exception NoEntityInterface
```

```
    {
        string   reason;
        string   details;
        EntityID entity;
    };

    // Thrown by objects that do not implement "optional" methods
    exception OptionNotImplemented
    {
        string   reason;
        string   details;
    };

    // Thrown when a request is made using data that is "old"
    exception OutOfDate
    {
        string    reason;
        string    details;
        Timestamp newTime;
    };

    // Thrown when a request cannot be satisfied because a
    // system component is not running or is otherwise unavailable
    exception ServiceNotAvailable
    {
        string reason;
        string details;
    };

    // Thrown when a request has been made for data that is
```

```
    // temporarily unavailable (due to loss of network connection,
    // for example)
    exception DataNotAvailable
    {
        string   reason;
        string   details;
    };

    // Thrown when a request has been made for data that does
    // not exist (note: use the NoSuchEntity exception if the missing
    // data corresponds to an EntityID)
    exception DataNotFound
    {
        string   reason;
        string   details;
    };

    // Thrown when a request has been made that is unable to complete
    // because data cannot be written permanently
    exception DataWriteError
    {
        string   reason;
        string   details;
    };

    // Thrown when a request has been made to use data that
    // is not of the valid form or range
    exception DataValidationError
    {
        string   reason;
```

```
        string   details;
    };

    // Any IEN interface that supports a ping() method should be
    // derived from Pingable
    // (This is derived from the NRTDS.)
    interface Pingable
    {
        void ping();

        // The "debug ping" method provides a back door for triggering
        // implementation-specific behavior.  It is intentionally left
        // undefined, and should only be used for debugging and
        // diagnostic purposes.  It allows developers to add special
        // methods to objects without modifying the interfaces.
        //
        // 'auth' should contain authentication information from the
        // caller (or be empty), and 'value' can be whatever
        //
        // The caller of the function needs to know what to do with
        // the returned 'any' value.
        any debug_ping(
            in any auth,
            in any value
        ) raises (
            OptionNotImplemented
        );
    };

    // All objects that represent "entities" should implement
```

```
    // the Entity interface
    interface Entity: Pingable
    {
        readonly attribute CorridorID    corridor;  // zero means "local corridor"
        readonly attribute SiteID        site;      // zero means "local site"
        readonly attribute octet         system;    // zero means "local system"
        readonly attribute octet         type;      // value from EntityType
enumeration
        readonly attribute EntityNumber  number;    // identification number

        readonly attribute EntityID      eid;       // full EntityID structure

        readonly attribute Identifier    name;      // brief name
        readonly attribute Description   desc;      // verbose description
    };
    typedef sequence<Entity> EntitySeq;

    // This exception is thrown by iterator interfaces' max_left()
    // method when the number of remaining elements is unknown
    exception UnknownMaxLeft
    {
        string reason;
        string details;
    };

    // Iterator for a set of EntityID structures
    interface EntityIDIterator
    {
        // Return true and the next N (or fewer) elements,
        // or return false if no more elements are available
```

```
        boolean next_n(
            in  unsigned long n,
            out EntityIDSeq   eids
        );

        // Return the number of elements that have not yet been
        // returned by the iterator
        unsigned long max_left()
            raises (UnknownMaxLeft);

        // Release the iterator
        void destroy();
    };

    // Iterator for a set of Entity object interfaces
    interface EntityIterator
    {
        // Return true and the next N (or fewer) elements,
        // or return false if no more elements are available
        boolean next_n(
            in  unsigned long n,
            out EntitySeq     entities
        );

        // Return the number of elements that have not yet been
        // returned by the iterator
        unsigned long max_left()
            raises (UnknownMaxLeft);

        // Release the iterator
```

```
        void destroy();
    };

    // All objects that "manage" a set of entities should inherit from
    // the EntityManager interface
    interface EntityManager: Pingable
    {
        // Return IDs of all entities of given type
        // (use ENTITYTYPE_NOT_AVAILABLE to get all entities)
        void listEntityIDs(
            in  EntityType       type,
            in  unsigned long    how_many,
            out EntityIDSeq      eids,
            out EntityIDIterator itr
        );

        // Return interface to given entity
        Entity getEntity(
            in EntityID eid
        ) raises (
            NoSuchEntity,
            NoEntityInterface,
            OptionNotImplemented
        );

        // Return interface pointers to all entities of given type
        // (use ENTITYTYPE_NOT_AVAILABLE for all entities)
        void listEntities(
            in  EntityType       type,
            in  unsigned long    how_many,
```

```
            out EntitySeq        entities,
            out EntityIterator   itr
        ) raises (
            NoEntityInterface,
            OptionNotImplemented
        );
    };

    // Service Context ID for implicit IEN-related information
    // in GIOP messages; this value has been allocated by the OMG.
    //
    // (If additional Service Context IDs are ever needed, it is
    // recommended that the range 0x49454E00-0x49454E0F be allocated.)
    const unsigned long ServiceContextID = 0x49454E00;  // "IEN\0"

};

#endif

//--------------------------------------------------------------------------
// Copyright 2001, 2009 County of Los Angeles. All Rights Reserved.
//--------------------------------------------------------------------------
```

5.1.2 IENRTData.idl

```
//------------------------------------------------------------------------
// Copyright 2009 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
//------------------------------------------------------------------------
// $Id: IENRTData.idl,v 1.6 2009/05/21 17:08:07 build Exp $
//
//------------------------------------------------------------------------

#ifndef IENRTDATA_IDL
#define IENRTDATA_IDL

#pragma prefix "transcore.com"

/// CORBA type definitions for IEN Real-time Data Distribution subsystem.
/// For MCS, only the DeviceType enumeration is still used.  The other types
/// defined here are obsolete.
module IENRTData
{
    const string IDLFileID = "$Id: IENRTData.idl,v 1.5 2009/01/30 14:15:26 mayoj 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,
        DT_DETECTOR_STATION
    };

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

};

#endif

//------------------------------------------------------------------------
// Copyright 2009 County of Los Angeles. All Rights Reserved.
//------------------------------------------------------------------------
```

5.1.3 TCS.idl

```
// ------------------------------------------------------------------------

// Copyright 2009 County of Los Angeles. All Rights Reserved.

//

// Developed by TransCore

// ------------------------------------------------------------------------

// $Id: tcs.idl,v 1.7 2009/01/23 20:52:25 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
};

/// 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 2009 County of Los Angeles. All Rights Reserved.
//---------------------------------------------------------------------------
```

5.1.4 IENTCSData.idl

```
//------------------------------------------------------------------------
// Copyright 2001, 2009 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
//------------------------------------------------------------------------
// $Id: IENTCSData.idl,v 1.16 2009/04/23 21:14:33 mayoj Exp $
//
// CORBA type definitions for IEN Traffic Control System (TCS)
// Command/Data Interface (CDI).
//
// All traffic control systems will not support all of these elements.
// For enumerated types, an "OTHER_NO_ADDITIONAL" value signifies that
// the TCS cannot translate its elements to a meaningful form.  For
// integer types, any value less than 0 generally indicates an unsupported
// data element.
//------------------------------------------------------------------------

#ifndef IENTCSDATA_IDL
#define IENTCSDATA_IDL

#include "tcs.idl"
#include "IENRTData.idl"

#pragma prefix "transcore.com"
module IENTCSData
{
    const string IDLFileID = "$Id: IENTCSData.idl,v 1.16 2009/04/23 21:14:33 mayoj Exp
$";
```

```
/// Types of device data to request from CDI.
   enum IEN_EventType
   {
       IEN_COMMANDRETURN,
       IEN_INTERSECTIONRTSTATUS,
       IEN_INTERSECTIONRTSUMMARY,
       IEN_PHASE_STATEDATA,
       IEN_PEDPHASE_STATEDATA,
       IEN_VEHCALL_STATEDATA,
       IEN_LASTCYCLE_PHASEDATA,
       IEN_DETECTORSTATE,
       IEN_INTERSECTIONINFO,
       IEN_DETECTORINFO,
       IEN_TP_PHASEDATA,
       IEN_SECTIONINFO,
       IEN_SECTIONSTATE
   };

/// Status information returned from commands.
enum IEN_CommandReturns {
     IEN_COMMAND_OK,
     ERR_CORRIDOR_BROKEN,
     ERR_SITE_BROKEN,
     ERR_CDISYS_BROKEN,
     ERR_CDISYS_UNAVL,
     ERR_INVALID_USER,
     ERR_INVALID_SITE,
     ERR_INVALID_CORRIDOR,
     ERR_INVALID_CDISYS,
     ERR_INVALID_CMDCODE,
```

```
        ERR_INVALID_CMDPARA,
        ERR_INVALID_DEVTYPE,
        ERR_INVALID_DEVNUM
    };

// INTERSECTION DATA

    // Timing plan selection modes reported for intersection controllers.
    enum SignalControlMode
    {
        ISC_OTHER_NO_ADDITIONAL,            // A mode other than those here
        ISC_OTHER_ADDITIONAL,               // Deprecated
        ISC_FREE,                           // free
        ISC_FIXED_TIME,                     // Fixed length phases
        ISC_TIME_BASE_COORDINATION,         // Coordinated clock-based plan
        ISC_ACTUATED,                       // Fully actuated ctrl (like free)
        ISC_SEMI_ACTUATED,                  // Semi-actuated (redundant with
                                            // time based coordination,
                                            // deprecated
        ISC_CRITICAL_INTERSECTION_CONTROL,  // Split adjustment based on traffic
                                            // at a critical intersection
        ISC_TRAFFIC_RESPONSIVE,             // Traffic responsive plan selection
        ISC_ADAPTIVE,                       // Using an adaptive algorithm
        ISC_TRANSITION,                     // Transition between plans
        ISC_EXTERNAL,                       // Plan from external system
        ISC_ATCS,                           // Special LADOT adaptive mode
        ISC_UNKNOWN
    };

    enum IntersectionSignalState
```

```
{
    ISS_OTHER_NO_ADDITIONAL,            // Obsolete, do not use
    ISS_OTHER_ADDITIONAL,               // Obsolete, do not use
    NORMAL_OPERATION,                   // Intersection operating normally
    FLASH,                              // Intersection is flashing
    PREEMPTION,                         // Preemption input active
    CONFLICT_FLASH,                     // Flashing due to conflict monitor
    FAILED,                             // Central system has failed the
                                        // controller
    ISS_UNKNOWN                         // State unknown
};

enum ControllerResponseState
{
    RESPONDING_OTHER_NO_ADDITIONAL,     // Obsolete, do not use
    RESPONDING_OTHER_ADDITIONAL,        // Obsolete, do not use
    RESPONDING,                         // Responded to last comm attempt
    NOT_RESPONDING,                     // No response to last comm attempt
    UNKNOWN                             // Response state unknown
};

enum PreemptType
{
    PREEMPT_OTHER_NO_ADDITIONAL,        // Preempt of type other than those
                                        // in this enumeration
    PREEMPT_OTHER_ADDITIONAL,           // Obsolete, do not use
    NO_PREEMPT,                         // No preemption in effect
    GENERAL_PREEMPT,                    // General preemption
    BRIDGE_PREEMPT,                     // Preemption by bridge
    EV_PREEMPT,                         // Preemption for emergency vehicle
```

```
        LRT_PREEMPT,                        // Preempt for light rail train
        RR_PREEMPT,                         // Railroad preempt
        PREEMPT_UNKNOWN                     // Preempt of unknown type
    };

    enum CommState
    {
        COMM_OTHER_NO_ADDITIONAL,           // Obsolete, do not use
        COMM_OTHER_ADDITIONAL,              // Obsolete, do not use
        COMM_GOOD,                          // Comm from central to ctrlr good
                                            // enough for central
        COMM_BAD,                           // Comm from central to ctrlr too
                                            // bad to use reliably
        COMM_UNKNOWN                        // Central can't determine comm
                                            // state to ctrlr, due to comm eqpmt
                                            // failure or ctrlr offline
    };

    /** Controller alarm bitmasks */
    const short NO_ALARM                = 0x00;
    const short CONFLICT_FLASH_ALARM    = 0x01;
    const short CABINET_DOOR_OPEN_ALARM = 0x02;
    const short TRANSITION_ALARM        = 0x04;
    const short INTERNAL_ERROR_ALARM    = 0x08;
    const short FLASH_ALARM             = 0x10;

// DETECTOR DATA

    /// Operational status of a detector.
    enum DetectorStatus
```

```
{
    /// Other state; no additional details available
    DETECTOR_OTHER_NO_ADDITIONAL,

    /// Other state; additional details available through
    /// device-specific interface
    DETECTOR_OTHER_ADDITIONAL,

    /// Enabled but not working due to hardware or comm failure
    DETECTOR_FAILED,

    /// working
    DETECTOR_OPERATIONAL,

    /// intentionally disabled
    DETECTOR_OFF,

    /// Detector state unknown due to communication problems, system
    /// configuration, or other central system problems.
    DETECTOR_UNKNOWN
};

/// Usage of the detector in the traffic system.
enum DetectorClass
{
    DC_OTHER_NO_ADDITIONAL,
    DC_OTHER_ADDITIONAL,
    DC_STOP_BAR,
    DC_SYSTEM,
    DC_PEDESTRIAN,
```

```
        DC_ADAPTIVE,
        DC_CALL,
        DC_EXTENSION,
        DC_MAINLINE,
        DC_REVERSIBLE_LANE,
        DC_RAMP_DEMAND,
        DC_RAMP_MERGE,
        DC_RAMP_PASSAGE,
        DC_RAMP_QUEUE,
        DC_UNKNOWN
    };

    /// Vehicle detection technique used by the detector.
    enum DetectorType
    {
        DT_OTHER_NO_ADDITIONAL,
        DT_OTHER_ADDITIONAL,
        DT_INDUCTIVE_LOOP,
        DT_MAGNETIC,
        DT_MAGNETOMETERS,
        DT_PRESSURE_CELLS,
        DT_MICROWAVE_RADAR,
        DT_ULTRASONIC,
        DT_VIDEO_IMAGE,
        DT_LASER,
        DT_INFRARED,
        DT_ROAD_TUBE,
        DT_UNKNOWN
    };
```

```
    /// Direction of traffic flow on a roadway or past a detector.
       enum Direction
    {
        EastBound,
        WestBound,
        SouthBound,
        NorthBound,
        SouthEastBound,
        SouthWestBound,
        NorthEastBound,
        NorthWestBound,
        InBound,
        OutBound,
        None,
        East_West,              // Bi-directional east & west
        North_South,            // Bi-directional north & sourth
        NE_SW,                  // Bi-directional northest and southwest
        NW_SE,                  // Bi-directional northwest and southeast
        InBound_and_Outbound,   // Bi-directional in and outbound
        Other                   // When none of the above will do . . .
      };
};

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, 2009 County of Los Angeles. All Rights Reserved.
//------------------------------------------------------------------------
```

5.1.5 MCSData.idl

```
// ------------------------------------------------------------------------
// Copyright 2009 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
// ------------------------------------------------------------------------
// $Id: mcsdata.idl,v 1.2 2009/02/18 16:41:05 mayoj Exp $
//
// Data definitions for MCS interfaces
// ------------------------------------------------------------------------

#ifndef MCSDATA_IDL
#define MCSDATA_IDL

#pragma prefix "transcore.com"



#include "IENRTData.idl"
#include "IENTCSData.idl"



module MCSDATA
{
       ///NEW: 32 bit device identifier instead of 16 bit.
    typedef long  MCSDeviceID;
    typedef sequence<MCSDeviceID> MCSDeviceIDList;

       ///NEW: 32 bit device identifier instead of 16 bit.
    ///System ID moved into SiteUpdate in MCSDataInterface.idl.
       struct MCSDevice
```

```
	{
		IENRTData::DeviceType type;
		MCSDeviceID id;
	};

	///NEW: 32 bit device identifier instead of 16 bit.
	typedef sequence<MCSDevice> MCSDeviceList;

	  /// A sequence of device event types
    typedef sequence<IENTCSData::IEN_EventType> EventTypeList;



};

#endif

//----------------------------------------------------------------------------
// Copyright 2009 County of Los Angeles. All Rights Reserved.
//----------------------------------------------------------------------------
```

5.1.6 MCSDataInterface.idl

```
//-----------------------------------------------------------------------------
// Copyright 2009 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
//-----------------------------------------------------------------------------
// $Id: MCSDataInterface.idl,v 1.13 2009/06/16 14:31:58 build Exp $
//-----------------------------------------------------------------------------

#ifndef MCS_DATA_INTERFACE_IDL
#define MCS_DATA_INTERFACE_IDL

#include "IEN.idl"
#include "IENTCSData.idl"
#include "tcs.idl"
#include "mcsdata.idl"

module MCSDataInterface {
     const short majorVersion = 3;
     const short minorVersion = 0;
     const short revision     = 0;

    ///Configuration information for a controller.
    struct IntersectionInfo
    {
        ///ID of the controller.
        MCSDATA::MCSDeviceID id;

        ///The EntityID of the section to which this intersection belongs. -1 if
        /// it is not a member of a section.
```

```
        long     sectionID;

        ///Seconds between poll attempts to the intersection controller.
        short    secondsBetweenPollAttempts;

        ///The name of the cross street of the intersection.
        string   crossStreet;

        ///The name of the main street of the intersection.
        string   mainStreet;

        ///The direction of traffic flow on the main street.
        IENTCSData::Direction mainStreetDirection;

        ///Latitude of the controller in micro degrees.  Positive if north of
        ///equator, negative if sourth.
        long     latitude;

        ///Longitude of the controller in micro degrees.  Positive if east of
        ///prime meridian, negative if west.
        long     longitude;

        ///Name describing the type of controller.
        string   controllerType;

        ///Brief description of where this controller is. Usually a main street
        ///@ cross street specification.
        string   description;
    };
```

```
typedef sequence<IntersectionInfo> IntersectionInfoSeq;

///Summary of real-time controller status.  Contains all data necessary for
///display of intersection status on the map.
struct IntersectionRTSummary
{
    ///Id of the controller.
    MCSDATA::MCSDeviceID                    id;

    ///Timing plan selection mode in use by the controller.
    IENTCSData::SignalControlMode           controlMode;

    ///Operational state of the controller.
    IENTCSData::IntersectionSignalState     signalState;

    ///Controller response state, RESPONDING if controller responded to last
    ///communication attempt, NOT_RESPONDING if not, UNKNOWN if system
    ///problem prevents communication.
    IENTCSData::ControllerResponseState     responseState;

    ///If preemption in effect, cause for preemption. NO_PREEMPT if
    ///preemption not in effect.
    IENTCSData::PreemptType                 preemptType;

    ///A bitmask representing one or more active alarms for the controller.
    long                                    alarms;

    ///True if main street green is active, false if inactive or unknown.
    boolean                                 isMainStreetGreen;
```

```
        ///Overall state of communication from central to the controller,
        ///COMM_GOOD if comm mostly functioning as desired, COMM_BAD if quality
        ///of comm is too low for use by central, UNKNOWN if central problem
        ///prevents communication.
        IENTCSData::CommState                   commState;

        ///Numeric id of the timing plan.
        long                                    timingPlanID;

        ///Desired cycle length in seconds.
        long                                    desiredCycleLength;

        ///Desired offset in seconds.
        long                                    desiredOffset;

        ///Observed offset in seconds, -1 if unknown.
        long                                    actualOffset;
    };

    typedef sequence<IntersectionRTSummary> IntersectionRTSummarySeq;

    //Cycle and Reference counters.
    struct IntersectionRTStatus
    {
        //ID of the controller.
        MCSDATA::MCSDeviceID                    id;

        //Seconds since the start of current cycle.
        short                                   cycleCounter;
```

```
        //Cycle counter for controller with the same cycle length but offset 0
        short                                   referenceCycleCounter;
    };

    typedef sequence<IntersectionRTStatus> IntersectionRTStatusSeq;

    //Contains the ids of the phases that are currently green.
    struct PhaseStateData
    {
        //ID of the controller.
        MCSDATA::MCSDeviceID id;

        //Numeric IDs of phases that are currently green.  The sequence length
        //should be equal to the number of phases that are currently active. If
        //there are none then this sequence should contain one element with a
        //value of zero.
        IEN::ShortSeq activeGreens;
    };

    typedef sequence<PhaseStateData> PhaseStateDataSeq;

    //Contains the ids of the phases that are currently displaying walk signals.
    struct PedestrianPhaseState
    {
        //Id of the controller
        MCSDATA::MCSDeviceID id;

        //Numeric IDs of phases that are currently displaying a walk signal.
        //The sequence length should be equal to the number of phases in walk.
        //If there are none then this sequence should contain one element with a
```

```
        //value of zero.
        IEN::ShortSeq activePeds;
    };

    typedef sequence<PedestrianPhaseState> PedestrianPhaseStateSeq;

    //Phases with active vehicle calls in a controller.
    struct VehicleCallState
    {
        //ID of the controller.
        MCSDATA::MCSDeviceID id;

        //Numeric IDs of phases with active vehicle calls.  The length of the
        //sequence should be the number of phases that have active vehicle
        //calls.  If there are none then the length should be one with a single
        //phase value of zero.
        IEN::ByteSeq phasesWithCalls;
    };


    typedef sequence<VehicleCallState> VehicleCallStateSeq;

    //Numeric identifier and phase time in seconds for a phase.
    struct PhaseTimeDescription
    {
        //Numeric identifier for a phase.
        short phaseId;

        //Number of seconds that the phase was or should be active.
        short phaseTime;
```

```
};

typedef sequence<PhaseTimeDescription> PhaseTimeDescriptionSeq;

//The green times for all the phases that were active in the controller
//during the just completed cycle.
struct LastCyclePhaseData
{
    //ID of the controller.
    MCSDATA::MCSDeviceID     id;

    //The total time of all the active phases;
    long                     totalPhaseTime;

    //Green times for phases that were active in the controller's last
    //completed cycle.  The length of the sequence should be the number of
    //phases that were active in the last cycle. Each element contains the
    //ID of an active phase and the total time in seconds that the phase was
    //active in the last completed cycle. If the totalPhaseTime field is
    //zero then the length of this sequence should be zero as well.
    PhaseTimeDescriptionSeq  greenTimes;
};

typedef sequence<LastCyclePhaseData> LastCyclePhaseDataSeq;

//Planned phase times for active phases in the current timing plan.
struct TpPhaseData
{
    //Id of the controller.
    MCSDATA::MCSDeviceID     id;
```

```
        //Planned phase times in seconds for the current timing plan in the
        //controller.  The length of the sequence should be the number of phases
        //active in the timing plan.  Each element in the sequence should
        //contain the phase ID and the planned phase time in seconds.  The
        //sequence length may be zero if the controller is currently running
        //free or is in flash.
        PhaseTimeDescriptionSeq  plannedPhaseTimes;
    };

    typedef sequence<TpPhaseData> TpPhaseDataSeq;

    ///Configuration information for a system detector.
    struct DetectorInfo
    {
        ///Id of the detector.
        MCSDATA::MCSDeviceID       id;

        ///Averaging period in seconds used to create averaged detector data. If
        ///averaged data is not available, should be -1.
        long                       averagingPeriod;

        ///Usage of the detector in the traffic system.
        IENTCSData::DetectorClass  detectorClass;

        /// Vehicle detection technique used by the detector.
        IENTCSData::DetectorType   detectorType;

        /// Direction of traffic flow past the detector.
        IENTCSData::Direction      detectorDirection;
```

```
    //Lane number for traffic passing the detector. The innermost lane
    //on the roadway is lane 1, the next lane to the right is lane 2, etc.
    octet                       laneNumber;

    //Name of the roadway that contains the detector.
    string                      roadName;

    //Weighting factor(k) for volume+weighted occupancy calculations.
    float                       weightingFactor;
};

typedef sequence<DetectorInfo> DetectorInfoSeq;

/// Most recent state information from the detector.  If the  status field
/// is any value other than DETECTOR_OPERATIONAL the volume, occupancy,
/// and speed fields should not be used.
struct DetectorState
{
    ///Id of the detector.
    MCSDATA::MCSDeviceID       id;

    ///The time of the updload in HHMMSS format: hours (0-23 for midnight to
    ///11 p.m.) * 10,000 | minutes (0-59) * 100 | seconds (0-59)
    IEN::HHMMSS                lastUpdateTime;

    ///The date of the update in YYYYMMDD format: year (AD) * 10,0000 |
    ///month (1-12) | day of month (1-31).
    IEN::YYYYMMDD              lastUpdateDate;
```

```
        ///Volume reported by the detector in vehicles per hour.
        long                        volume;

        /// Average volume (-1 if not available).
        long                        avgVolume;

        ///Volume in veh. per hour + weighted occupancy (-1 if not available).
        long                        volumePlusWeightingFactor;

        ///Avg volume + weighted occupancy. (-1 if not available).
        long                        avgVolumePlusWeightingFactor;

        ///Latest known status of the detector.  If the field is a value other
        ///than DETECTOR_OPERATIONAL then the volume, occupancy, and speed
        ///fields should be set to -1.
        IENTCSData::DetectorStatus status;

        ///Speed in miles per hour.  -1 if not available.
        short                       speed;

        ///Average speed over averaging period.  -1 if not available.
        short                       avgSpeed;

        ///Occupancy percentage.
        short                       occupancy;

        ///Average occupancy.  -1 if not available.
        short                       avgOccupancy;
    };
```

```
typedef sequence<DetectorState> DetectorStateSeq;


struct DetectorStationInfo
{
     //32 bit device id
     MCSDATA::MCSDeviceID  id;
     //
     long                  freewayId;
     //Latitude
     long                  latitude;
     //Longitude
     long                  longitude;
     //Post Mile Marker
     float                 postMileMarker;
     //Direction of traffic flow
     IENTCSData::Direction detectorDirection;
     //Number of lanes
     octet                 numberOfLanes;
     //Name of roadway
     string                roadwayName;
     //Name of cross street
     string                crossStreet;
     //A description of this station
     string                description;
     //The source of the Update PeMS or RIITS
     string source;
};


typedef sequence<DetectorStationInfo> DetectorStationInfoSeq;
```

```
struct DetectorStationState
{
    //32 bit id of the device
    MCSDATA::MCSDeviceID       id;
    //Status of the detector station. (OK or failed)
    IENTCSData::DetectorStatus status;
    //Volume
    float                      volume;
    //Occupancy
    float                      occupancy;
    //Speed
    float                      speed;
    //HOV Speed
    float                      hovSpeed;
    //The source of the Update either PeMS or RIITS
    string source;
};

typedef sequence<DetectorStationState> DetectorStationStateSeq;

/// Configuration information for a section.
struct SectionInfo
{
    //ID of section.
    MCSDATA::MCSDeviceID id;

    //The IDs of controllers that belong to this section.
    IEN::LongSeq         intersections;
};
```

```
typedef sequence<SectionInfo> SectionInfoSeq;

///Current state of the section.
struct SectionState
{
    //Id of the Section.
    MCSDATA::MCSDeviceID            id;

    ///Control mode used to select the timing plans of controllers following
    ///the section.
    IENTCSData::SignalControlMode   sectionControlMode;

    //Timing plan of the controllers that follow this Section.
    short                           planID;
};

typedef sequence<SectionState> SectionStateSeq;

/// Device configuration data.   Contains IntersectionInfo, DetectorInfo,
/// and SectionInfo sequences for a given site and system.
struct DeviceConfigGroup
{
    /// Time update was created by the CDI.
    IEN::HHMMSS            timeOfDistribution;

    /// Configuration information for intersections.
    IntersectionInfoSeq    intersections;

    /// Configuration information for detectors.
    DetectorInfoSeq        detectors;
```

```
    /// Configuration information for sections.
    SectionInfoSeq         sections;

    /// Configuration information for detector stations
    DetectorStationInfoSeq stations;
};

typedef sequence<DeviceConfigGroup>  DeviceConfigBatch;

/// Slowly changing device state data.  Contains the IntersectionRTSummary,
/// DetectorState, and SectionState sequences for a given site and system.
struct DeviceStateGroup
{
    /// Time update was created by the CDI.
    IEN::HHMMSS              timeOfDistribution;

    /// Intersection summary information (suitable for map).
    IntersectionRTSummarySeq intersectionStates;

    /// Detector state information (last set of uploaded data).
    DetectorStateSeq         detectorStates;

    /// Section state information (control mode and timing plan in use).
    SectionStateSeq          sectionStates;

    /// Detector Station states
    DetectorStationStateSeq  stationStates;
};
```

```
typedef sequence<DeviceStateGroup>  DeviceStateBatch;

/// Intersection data for a given site and system, mostly rapidly changing.
/// These data types are only delivered when a user interface program
/// subscribes to them.
struct RealTimeUpdateGroup
{
    /// Time update was created by the CDI.
    IEN::HHMMSS                 timeOfDistribution;

    /// Intersection cycle counters and master cycle counters.
    IntersectionRTStatusSeq     intersectionRTStatuses;

    /// Active green phases for intersections.
    PhaseStateDataSeq           intersectionPhaseStates;

    /// Active pedestrian walk signals for intersections.
    PedestrianPhaseStateSeq     pedestrianPhaseStates;

    /// Active vehicle calls for intersections.
    VehicleCallStateSeq         vehicleCallStates;

    /// Phase times from completed cycles for intersections.
    LastCyclePhaseDataSeq       lastCycles;

    /// Planned phase times for intersections.
    TpPhaseDataSeq              maximumPhaseTimes;
};

typedef sequence<RealTimeUpdateGroup>  RealTimeUpdateBatch;
```

```
///Contains Info,State, Realtime update batches.
struct SiteUpdate
{
    /// Display ID of site of origin for the update.  Filled in by the
    /// site server, not the CDI.
    short               site;

    /// Display ID of system of origin for the update.   Filled in by the
    /// site server, not the CDI.
    short               system;

    /// Size of the data contained in the update in bytes.  Filled in by the
    /// site server after getting the update from the CDI.
    long                size_of_data;

    /// Requested configuration information for devices on the TCS.
    DeviceConfigGroup   configUpdates;

    /// Requested device state information for the TCS.
    DeviceStateGroup    stateUpdates;

    /// Requested intersection real-time state information for the TCS.
    RealTimeUpdateGroup realTimeUpdates;
};

typedef sequence<SiteUpdate> SiteUpdateBatch;

/// Event types returned for a given device type
struct MCSDeviceDataTypes
```

```
{
    /// Type of device that the CDI supports
    IENRTData::DeviceType  type;

    /// Types of events that a site server may request for the device type.
    MCSDATA::EventTypeList eventTypes;
};

/// List of data types for all supported devices.
typedef sequence<MCSDeviceDataTypes> MCSDeviceDataTypeList;

///Specifies a MCSDevice and the types of data needed and if should look for
///changed data only.
struct MCSDeviceRequest
{
    /// Device type and 32 bit device ID.
    MCSDATA::MCSDevice      device;

    /// List of event types to request for the device.
    MCSDATA::EventTypeList  eventTypes;

    /// True if requesting changes since the last time the device's data
    /// was requested, false if the CDI should return all data for the
    /// device regardless of what data was last reported by the CDI.
    boolean                 changedOnly;
};

/// List of MCSDeviceRequest elements
typedef sequence<MCSDeviceRequest> MCSDeviceRequestList;
```

```
/// Status of a failed data request.  Used to report failed data requests to
/// site server.
enum ReasonForRequestFailure
{
    /// requested event type not supported
    EVENT_TYPE_NOT_SUPPORTED,

    /// access to data for device denied
    ACCESS_DENIED,

    /// requested device does not exist
    NO_DEVICE,

    /// other unspecified failure
    OTHER
};

/// Report the failure of a request for a given event type from a given
/// device.
struct RequestFailureStatus
{
    /// Device for which request failed.
    MCSDATA::MCSDevice          device;

    /// Type of data requested.
    IENTCSData::IEN_EventType   ien_eventType;

    /// Reason data request failed.
    ReasonForRequestFailure     reason;
};
```

```
typedef sequence<RequestFailureStatus> RequestFailureStatusSeq;

/// Data access interface for new MCS CDI programs.  Replaces the
/// data access methods in DataAccessor, but not the configuration/lifecycle
/// methods and properties.
interface MCSDataAccessor : TCSData::DataAccessor {
    /// Get the event types supported for all device types
    /// for which this CDI returns data.
    /// @return       supported data types for all supported devices
    MCSDeviceDataTypeList getDeviceEventTypes( );

    ///Get a list of devices accessible to the IEN of the given types.
    ///
    ///@param types device types in which the caller is interested
    ///
    ///@return The 32 bit version of the devices currently configured to
    ///        report to the IEN.
    MCSDATA::MCSDeviceList getAvailableDevices32(
        in TCS::DeviceTypeList types);

    /// Request TCS data by device and event type from the CDI.  The CDI
    /// should return all the data that it can in the set_update structure.
    /// For those requests that the CDI cannot satisfy, it should report the
    /// device ID, type and reason the request failed in the failed_requests
    /// sequence.
    ///
    /// @param device_requests    List of devices and data types requested
    /// @param site_update        Requested TCS data for those devices and
    ///                           event types that the CDI can satisfy.  The
```

```
        ///                             site server fills in the site, system, and
        ///                             size_of_data fields.  The CDI fills in the
        ///                             configUpdates, stateUpdates, and
        ///                             realTimeUpdates sequences.  If there is no
        ///                             data to report, the CDI may set the length
        ///                             of these sequences to zero.
        ///                             No data should be reported for failed
        ///                             requests or for change-only requests for
        ///                             which the device's configuration or state
        ///                             hasn't changed since the last report.
        ///                             The CDI should report the reasons for
        ///                             failed requests in the failed_requests
        ///                             parameter.
        /// @param failed_requests     Reasons for failed requests. Should have
        ///                             one element for each request in
        ///                             device_requests for which there is an
        ///                             exceptional failure.  No data reported to
        ///                             change-only requests is not considered a
        ///                             failure and should not be reported here.
        void getDeviceUpdate32(in MCSDeviceRequestList device_requests,
                               inout SiteUpdate site_update,
                               out RequestFailureStatusSeq failed_requests)
                                 raises (TCS::SystemStatusException,TCS::Error);
    };

};

#endif

//----------------------------------------------------------------------------
```

```
// Copyright 2009 County of Los Angeles. All Rights Reserved.

//---------------------------------------------------------------------------
```

5.1.7 Tcscommand.idl

```
// ------------------------------------------------------------------------
// Copyright 2001, 2009 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
// ------------------------------------------------------------------------
// $Id: tcscommand.idl,v 1.5 2009/02/12 16:49:14 mayoj Exp $
//
// CORBA interface to TCS commands.
// ------------------------------------------------------------------------

#ifndef TCSCOMMAND_IDL
#define TCSCOMMAND_IDL

#include "tcs.idl"
#include "IENRTData.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 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 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 2009 County of Los Angeles. All Rights Reserved.
//---------------------------------------------------------------------------
```

5.1.8 Mcscommand.idl

```
// ------------------------------------------------------------------------
// Copyright 2008, 2009 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
// ------------------------------------------------------------------------
// $Id: mcscommand.idl,v 1.8 2009/05/18 18:20:02 mayoj Exp $//
// New CORBA interface to TCS commands for MCS.  Adds flash method to the
// existing CommandAccessor interface.
// ------------------------------------------------------------------------
#ifndef MCSCOMMAND_IDL
#define MCSCOMMAND_IDL

#include "tcscommand.idl"
#include "mcsdata.idl"

module MCSCommand
{
    /// Major version, minor version, and revision of this IDL interface
    const short majorVersion = 3;
    const short minorVersion = 0;
    const short revision     = 0;

    /// Result of sending a command to one device.
    struct CommandResult {
        /// Type and ID of commanded device.
        MCSDATA::MCSDevice device;

        /// True if command successful, false if not.
        boolean isSuccessful;
```

```
        /// Free-form string for reporting the reason the command failed.
        string message;
    };

    typedef sequence<CommandResult> CommandResultSequence;

    /// Thrown if devices are specified that
    /// are unknown to TCS or not fully configured
    exception MCSUnknownDevices
    {
        MCSDATA::MCSDeviceList unknowns;
    };

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

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

```
/// Thrown when a command is not supported by the CDI for one or more of the
/// requested devices.
exception MCSCommandNotSupported
{
    MCSDATA::MCSDeviceList devices;
};

/// Enables IEN programs to send commands to a traffic control system.
interface MCSCommandAccessor: TCSCommand::CommandAccessor
{
    ///Get a list of devices of the given types to which the IEN may give
    ///commands.
    ///
    ///@param types device types in which the caller is interested
    ///
    ///@return The 32 bit version of the devices that may currently accept
    ///        commands from the IEN
    MCSDATA::MCSDeviceList getAvailableDevices32(
        in TCS::DeviceTypeList types);


     /// Change CDI plan number using 32-bit device IDs.
    ///
    /// @param devices    List of devices for which to change the plan
    /// @param planNumber New plan number for the requested devices
    /// @return           Results for command to each device on list
    /// @throws           CommandsNotAccepted if
    CommandResultSequence setCDIPlan32(
        in MCSDATA::MCSDeviceList devices,
        in short                  planNumber
```

```
    ) raises (
        TCSCommand::CommandsNotAccepted,
        MCSInvalidPlanNumber,
        MCSUnknownDevices,
        MCSCommandNotSupported,
        TCS::SystemStatusException,
        TCS::Error
    );

    /// Change operational mode of specified devices using 32-bit device
    /// IDs.
    ///
    /// @param devices    List of devices for which to change the mode with
    ///                   32-bit device IDs.
    /// @param newMode    New operational mode for the devices on the list
    /// @return           Results for command to each device on list
    CommandResultSequence changeMode32(
        in MCSDATA::MCSDeviceList devices,
        in TCS::Mode              newMode
        ) raises (
            TCSCommand::CommandsNotAccepted,
            MCSInvalidMode,
            MCSUnknownDevices,
            MCSCommandNotSupported,
            TCS::SystemStatusException,
            TCS::Error
        );

    /// Release IEN control of a list of TCS devices using 32-bit device
    /// IDs.
```

```
        ///
        /// @param devices    List of devices for which to release IEN control
        ///                   with 32-bit device IDs
        /// @return           Results for command to each device on list
        CommandResultSequence releaseControl32(
            in MCSDATA::MCSDeviceList devices
            ) raises (
                MCSUnknownDevices,
                MCSCommandNotSupported,
                TCS::SystemStatusException,
                TCS::Error
            );

        /// Put MCS devices into flashing operation.  Remove from flash by
        /// calling releaseControl32.
        ///
        /// @param devices  List of devices to put into flash, with 32-bit
        ///                 device IDs.
        /// @return         Results for flash command to each device on list
        CommandResultSequence flash32(
            in MCSDATA::MCSDeviceList devices
            ) raises (
                TCSCommand::CommandsNotAccepted,
                MCSCommandNotSupported,
                MCSUnknownDevices,
                TCS::SystemStatusException,
                TCS::Error
            );    };

};
```

```
#endif


// ------------------------------------------------------------------------
// Copyright 2008, 2009 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
// ------------------------------------------------------------------------
```

5.2 APPENDIX B: EXAMPLE CDI MAIN CLASS

```
// ---------------------------------------------------------------------------
// Copyright (c) 2009 Transcore ITS, LLC. All rights Reserved.
//
// PROPRIETARY
//
// THIS SOURCE CODE IS THE PROPERY OF TRANSCORE. IT MAY BE USED BY
// RECIPIENT ONLY FOR THE PURPOSE FOR WHICH IT WAS TRANSMITTED AND MUST
// BE RETURNED UPON REQUEST OR WHEN NO NEEDED BY RECIPIENT. IT MAY NOT
// BE COPIED OR COMMUNICATED WITHOUT THE WRITTEN CONSENT OF TRANSCORE.
// ---------------------------------------------------------------------------
// File : $Id: IEN_CDI.java,v 1.23 2009/03/18 20:18:41 mayoj Exp $
// Author : welchj
// ---------------------------------------------------------------------------
package com.transcore.tcs.external.ien;

import java.util.*;

import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;
import org.omg.CORBA.ORBPackage.InvalidName;
import org.omg.CosNaming.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POAManagerPackage.AdapterInactive;
import org.omg.PortableServer.POAPackage.*;



/**
 * Main class for example IEN CDI.  Sets up the main CDI thread
 * and processes TCS data changes and command requests.
```

```
 *
 * @author Jay Welch
 * @author  $Author: mayoj $
 * @version $Revision: 1.23 $
 */
public class IEN_CDI implements Runnable {
    // How long to sleep between checking for ORB traffic, in milliseconds.
    private static final int IEN_Waiting_Period = 500;

    private org.omg.CORBA.Object data_factory_ = null;
    private org.omg.CORBA.Object command_factory_ = null;
    private TCS_CommandAccessorFactory tcs_command_factory_ = null;
    private TCS_DataAccessorFactory tcs_data_factory_ = null;

    private static final String site_and_system_id_ = "1:1";

    private static org.jacorb.orb.ORB orb = null;
    private static POA poa;

    private static final String siteServerName = "siteServer";
    private static final String orbArgs[] = {
        "-ORBInitRef",
        "NameService=corbaloc::" + siteServerName + ":14444/NameService"
        };

    /**
     * Constructor.  Creates the CORBA data accessor factory and command accessor
     * factory objects.
     *
     * @param system_interface
```

```
     *            ExternalSystemInterfaceBean Contains the properties that will
     *            be needed to have the IEN system contact this TCS for updates
     *            and command execution.
     *
     */
    public IEN_CDI() {
        tcs_command_factory_ = new TCS_CommandAccessorFactory(getPoa());
        tcs_data_factory_ = new TCS_DataAccessorFactory(getPoa());
    }

    /**
     * Get the CORBA ORB object for basic ORB functions.
     *
     * @return ORB object for basic ORB functions
     */
    public ORB getOrb() {
        if (orb == null) {
            orb = (org.jacorb.orb.ORB)ORB.init(orbArgs, (Properties) null);
        }
        return orb;
    }

    /**
     * Get the CORBA portable object adapter (POA) for server objects.
     *
     * @return POA object for use in creating server objects
     */
    public POA getPoa() {
        if (poa == null)
        {
```

```
            try
            {
                POA tmpPoa;
                tmpPoa = POAHelper.narrow(getOrb().resolve_initial_references(
                    "RootPOA"));
                tmpPoa.the_POAManager().activate();
                poa = tmpPoa;
            }
            catch (InvalidName ex)
            {
                // TODO Auto-generated catch block
                ex.printStackTrace();
            }
            catch (AdapterInactive ex)
            {
                // TODO Auto-generated catch block
                ex.printStackTrace();
            }
        }
        return poa;
    }

    /**
     * Main communication thread for the IEN CDI thread.  Activates the data and
     * command accessor factories, then enters the outer loop.
     *
     * The outer loop registers the data and command accessor factories in the
     * name service & logs success or failure.  If the names are registered
     * correctly, enters the inner loop.
     *
```

```
     * The performWork method checks to see that the naming service is still
     * available, then waits for ORB processing.  When performWork is
     * interrupted or the naming service becomes inaccessible, returns to the
     * outer loop to re-register the factory object's names in the naming
     * service.
     *
     * @see java.lang.Runnable#run()
     */
    public void run() {
        System.out.printf("%s %s","ien", "IEN Component has started running.");
        NamingContextExt naming_service = null;
        try {
            data_factory_ = getPoa()
                    .servant_to_reference(tcs_data_factory_);
            command_factory_ = getPoa().servant_to_reference(
                    tcs_command_factory_);

            while (!Thread.currentThread().isInterrupted()) {
                naming_service = resolveNamingContext();
                System.out.printf("%s %s","ien", "IEN NamingContext resolved.");
                if (bindFactories(naming_service)) {
                    System.out.printf("%s %s", "Informational",
                                    "All factories published. Waiting for IEN I/O.");
                    System.out.printf("%s %s", "ien",
                        "IEN Factories bound to the appropriate naming service.");

                    performWork(naming_service);

                    tcs_data_factory_.deactivateAccessors();
                    tcs_command_factory_.deactivateAccessors();
```

```
                } else {
                    Thread.sleep(5000);
                }
            }
        } catch (InterruptedException e) {
            // Just exit from the loop
        } catch (ServantNotActive e) {
            e.printStackTrace();
        } catch (WrongPolicy e) {
            e.printStackTrace();
        }
        cleanup(naming_service);

        System.out.printf("%s %s","ien", "IEN Component has stopped.");
    }

    /**
     * Wait for ORB operations until loop interrupted or naming service
     * reference becomes invalid (indicating the naming service has been
     * restarted).
     *
     * @param naming_service          reference to CORBA naming service
     *
     * @throws InterruptedException   If thread should terminate
     */
    private void performWork(NamingContextExt naming_service)
            throws InterruptedException
    {
        ORB orb = getOrb();
        try {
```

```
            while (!Thread.currentThread().isInterrupted()
                    && !naming_service._non_existent()) {
                if (orb.work_pending()) {
                    orb.perform_work();
                }
                Thread.sleep(IEN_Waiting_Period);
            }
        } catch (SystemException e) {
            System.out.printf("%s %s: %s", "Error",
                    "Exception occured while running the IEN interface",
                    e.getLocalizedMessage());
            e.printStackTrace();
        }
    }

    /**
     * Attempts to find the naming service where the command and data
     * factory names are to be published.  If not found, waits 10 seconds and
     * then tries again.
     *
     * @return NamingContextExt The naming context where the command and data
     *         accessor factories are to be published.
     *
     * @throws InterruptedException if the name service is not accessible or
     * another thread interrupts this one.
     */
    private NamingContextExt resolveNamingContext() throws InterruptedException {
        System.out.printf("%s %s","ien",
                "Resolving the IEN Site servers NameService");
        NamingContextExt naming_service = null;
```

```
String error = null;
int total_tries = 0;
ORB orb = getOrb();
while (!Thread.currentThread().isInterrupted()
        && naming_service == null) {
    String name_spec = "corbaloc:iiop:"
            + siteServerName + ":14444/NameService";
    total_tries += 1;
    try {
        System.out.printf("%s %s %s",
                "ien",
                "Resolving nameserver for the site server at ",
                siteServerName);
        org.omg.CORBA.Object obj = orb.string_to_object(name_spec);
        if (obj != null) {
            System.out.printf("%s %s",
                    "ien",
                    "ORB has resolved the initial reference "
                            + name_spec + " attempting narrow");
            naming_service = NamingContextExtHelper.narrow(obj);
        } else {
            error = "Unable to get initial reference to Name Service using "
                    + name_spec;
        }
    } catch (SystemException e) {
        error = "Unable to resolve naming service: "
                + e.getLocalizedMessage();
    } catch (Exception e) {
        error = "Unable to resolve naming service: "
                + e.getLocalizedMessage();
```

```
            }


            if (naming_service == null) {
                if (total_tries == 1 || total_tries % 600 == 0) {
                    System.out.printf("%s %s", "Error",
                                    "Unable to intialize CORBA Components.");
                }
                if (error != null) {
                    System.out.printf("%s %s","ien", error);
                } else {
                    System.out.printf("%s %s",
                            "ien",
                            "Unable to determine why the name service could not be "
                                    + "resolved.");
                }
                error = null;
                Thread.sleep(10000);
            }
        }
        if (naming_service != null) {
            System.out.printf("%s %s","ien", "Obtained naming service");
        } else {
            throw new InterruptedException();
        }
        return (naming_service);
    }


    /**
     * Binds the factories to the name service.
     *
```

```
     * @param naming_service
     *            NamingContextExt The naming context to bind to.
     *
     * @return boolean True indicates that the data and command interfaces have
     *         been bound to the given naming context false indicates that there
     *         was some problem binding the components to the name service
     */
    private boolean bindFactories(NamingContextExt naming_service) {
        String error = null;
        boolean bound = false;
        try {
            String just_site_id = site_and_system_id_ ;
            int sys_ndx = just_site_id.indexOf(":");
            if (sys_ndx != -1) {
                just_site_id = just_site_id.substring(0, sys_ndx);
            }
            naming_service.rebind(new NameComponent[] { new NameComponent(
                    "TCSCDIData" + site_and_system_id_, "Site"
                            + just_site_id) }, data_factory_);
            naming_service.rebind(new NameComponent[] { new NameComponent(
                    "TCSCDICmd" + site_and_system_id_, "Site"
                            + just_site_id) }, command_factory_);
            bound = true;
        } catch (Exception e) {
            error = "Unable to bind factories Exception: "
                    + e.getClass().getCanonicalName() + " message "
                    + e.getLocalizedMessage();
        }

        if (error != null) {
```

```
            System.out.printf("%s %s","ien", error);
        }
        return (bound);
    }

    /**
     * Clean up just before the CDI loop exits
     *
     * @param naming_service   CORBA naming service from which to remove the
     *                         CDI's names
     */
    private void cleanup(NamingContextExt naming_service)
    {
        if (data_factory_ != null) {
            tcs_data_factory_.deactivateAccessors();
            try {
                getPoa().deactivate_object(
                        getPoa().reference_to_id(data_factory_));
            } catch (Exception e) {
            }
        }

        if (command_factory_ != null) {
            tcs_command_factory_.deactivateAccessors();
            try {
                getPoa().deactivate_object(
                        getPoa().reference_to_id(command_factory_));
            } catch (Exception e) {
            }
        }
```

```
        if (naming_service != null) {
            unbindFactories(naming_service);
        }
    }

    /**
     * Unbinds the factories from the naming service.
     *
     * @param naming_service
     *            NamingContextExt The naming context to unbind from.
     *
     * @return boolean True indicates that the data and command interfaces have
     *         been unbound from the given naming context false indicates that
     *         there was some problem unbinding the components from the name
     *         service
     */
    private boolean unbindFactories(NamingContextExt naming_service) {
        String error = null;
        boolean unbound = false;
        try {
            naming_service.unbind(new NameComponent[] { new NameComponent(
                    "TCSCDIData" + site_and_system_id_, "Site"
                            + site_and_system_id_) });
            naming_service.unbind(new NameComponent[] { new NameComponent(
                    "TCSCDICmd" + site_and_system_id_, "Site"
                            + site_and_system_id_) });
            unbound = true;
        } catch (Exception e) {
            error = "Unable to unbind the factories from the naming service: "
                    + e.getClass().getCanonicalName() + " message "
```

```
                    + e.getLocalizedMessage();
        }


        if (error != null) {
            System.out.printf("%s %s","ien", error);
        }
        return (unbound);
    }
}
```

5.3 APPENDIX C: DIAGNOSTIC SETTINGS IN THE SITE SERVER PROGRAM

The Site Server program can be configured to write diagnostic output to a log file, which CDI Developers may find helpful when diagnosing problems with CDI programs. The configuration file for the Site Server (usually site.properties) contains an ien.loglevels property similar to the following:

```
ien.loglevels = info,warning,siteserver,cdihandler,sitedistributor,
```

This property governs the diagnostic output that the Site Server writes for data that it receives from a CDI. To change the quantity of CDI communication that the Site Server prints out, change the list of log levels defined in the value of the property. Multiple log levels can be set by naming them in a comma-separated list, as shown above. The Site Server should detect the changed settings shortly after the changes are saved to the configuration file and begin printing out the requested diagnostic data.

Table 5-1 below lists the diagnostic levels available with the Site Server (note that the level names are not case-sensitive):

Table 5-1: Diagnostic Levels for the Site Server

DIAGNOSTIC LEVEL	EXPLANATION
Fatal	An error that caused the Site Server to terminate. Cannot be disabled.
Error	Messages about recoverable errors that won't stop the Site Server. Cannot be disabled.
Info	Informational messages that are nice to know but not terribly important.
Warning	Warnings of possible problems.
Siteserver	Messages about naming service interactions and workstations connecting to and disconnecting from the Site Server.
CDIHandler	Interactions with Command/Data Interface (CDI) programs.
SiteDistributor	Distribution of data from the regional server to workstations.
UpdateCycle	Decisions about when to request change-only data vs. all data for devices from a CDI.
TopicManager	Creation, renewal, and termination of subscriptions.
TopicFilter	Transmission of real-time data to other sites when subscriptions exist.
SiteOutgoing	Transmission of data to the Regional Server and to workstation data services.
SiteIncoming	Distribution of data from the regional server to workstations.
Changes	Reports on the IDs of devices for which data has changed.
Command	Problems executing commands.
UpdateFailure	Failures getting data from version 3 CDIs.