Oracle® Communications Network Integrity

File Transfer and Parsing Guide

Release 7.3.2

E66040-01

May 2016

Oracle Communications Network Integrity File Transfer and Parsing Guide, Release 7.3.2

E66040-01

Contents

Preface

This guide describes Oracle Communications Network Integrity file transfer and parsing functionality.

Audience

This guide is intended for Network Integrity cartridge developers who want to either build or extend cartridges similar to the samples provided in this guide, and who want to use Network Integrity processors to transfer and parse files.

It is recommended that you be familiar with the following documents:

- *Network Integrity Concepts*
- *Network Integrity Developer's Guide*

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Document Revision History

The following table lists the revision history for this guide:

Version	Date	Description
E66040-01	May 2016	Initial release.

1 Overview

This chapter provides an overview of Oracle Communications Network Integrity file transfer and parsing functionality. This functionality is provided by processors and cartridges.

File Transfer and Parsing Processors

In some networks, the following entities can write files to the file system:

- devices
- users
- third-party applications

These files contain data that you can collect and model. Use Network Integrity to collect and model this data, to retrieve all the remote files and process them using processors. Network Integrity provides two processor types to help you develop cartridges, and then transfer and parse the required files. The processors are part of Oracle Communications Design Studio and include configuration options that help you create the cartridges that you need. This functionality applies to any domain.

The two processor types are:

- The file transfer processor transfers files from a remote device or EMS to a Network Integrity file system. This processor can also be used to access files on a local file system, which is shared between all the nodes in a Network Integrity cluster.
- The file parser processor parses, or interrogates, the contents of a file. The file parser can parse ASCII and XML files. See "The File Parser Processor" for more information.

Note: Files that cannot be parsed by the file parser processor can still be parsed by adding custom parsing code to a discover, import, or assimilation processor implementation.

Reference Cartridges

This guide provides two reference cartridges that serve as examples of creating a processor chain, which retrieves and parses ASCII files or XML files.

- The ASCII Reference Cartridge
- The XML Reference Cartridge

2 The File Transfer Processor

This chapter describes the file Oracle Communications Network Integrity file transfer processor.

About the File Transfer Processor

The file transfer processor is similar to other Network Integrity processors, with the following exceptions:

- The complete implementation is generated.
- It can be added to discovery, import, and assimilation actions.

File Transfer Output Parameters

The output parameter that the file transfer processor produces can then be used by the file parser processor as input for the parsing process.

The file transfer processor outputs a single parameter. This output parameter holds a collection of file objects, each of which points to the local version of the transferred file. This collection of files is used as input for a file parser processor, but can also be used by any type of processor in the action. The file collection output parameter can also be used as input to a For Each structure in the action to loop over the files individually.

The name of the output parameter is system-generated and based on the name of the file transfer processor. For example, the Sample File Transfer processor outputs sampleFileTransferFileCollection.

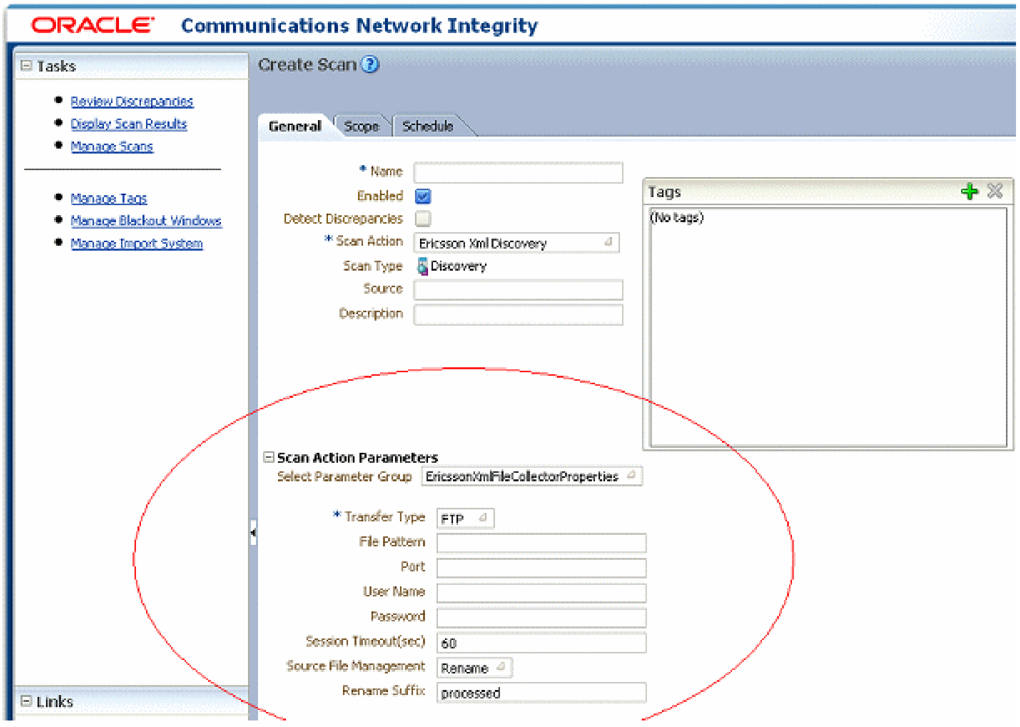
The type of the output parameter is always **java.util.Collection<java.io.File>**.

The Processor editor **Context Parameters** tab is read-only for file transfer processors, the **Usage** button shows which processors are using the output parameter.

Scan Parameter Groups

When a cartridge is deployed with scan parameter groups that are generated from a file transfer processor, they appear in the Network Integrity UI as scan parameters, as shown in Figure 2–1.

Figure 2–1 Scan Parameters in Network Integrity UI



* Name:
Enabled: ✓
Detect Discrepancies:
* Scan Action: Ericsson Xml Discovery
Scan Type: Discovery
Source:
Description:

Scan Action Parameters
Select Parameter Group: EricssonXmlFileCollectorProperties
* Transfer Type: FTP
File Pattern:
Port:
User Name:
Password:
Session Timeout(sec): 60
Source File Management: Rename
Rename Suffix: processed

Table 2–1 describes the default characteristics in the file transfer scan parameter group.

Table 2–1 Characteristics in the File Transfer Processor Scan Parameter Group

Characteristic Name	Default	Mandatory	Description
Transfer Type	FTP	Yes	Select how files should be transferred: FTP, SFTP, Local.
File Pattern	N/A	No	A pattern to match file names. The pattern supports wildcard characters. The supported wildcard characters are "*", "%", and "_". "*" and "%" represent a match of zero or more characters. "_" represents a match of any single character. Wildcard characters can be escaped with a back slash.
Port	N/A	No	The port used to connect to the remote server. The default is 21 for FTP, and 22 for SFTP.
User Name	N/A	No	The user name to connect to the remote location.
Password	N/A	No	The password to connect to the remote location.
Session Timeout	60	No	The amount of time in seconds before an idle connection is timed out. The valid range is from 1 to 3600.
Source File Management	Rename	No	Select the action to take on source files when the file transfer is complete. Options are: Delete, Rename, Nothing.

Table 2–1 (Cont.) Characteristics in the File Transfer Processor Scan Parameter Group

Characteristic Name	Default	Mandatory	Description
Rename Suffix	Processed	No	The suffix to add to the source file if the source file management characteristic has a value of Rename.

Note: Do not modify the characteristics listed in Table 2–1. You can create new characteristics in the generated scan parameter group, but the auto-generated characteristics must not be modified.

File Transfer Input Parameters

You can configure the file transfer processor to use Java objects that are available in the action context as input. This allows predecessor processors to programmatically control the behavior of the file transfer processor at run time.

The input parameters that the file transfer processor uses depend on the options configured in Oracle Communications Design Studio. For example, if the **Parameter Source** option is set to **Context Parameter** the file transfer processor requires an input parameter of the following type: **oracle.communications.sce.integrity.sdk.fileTransferCollector.FileTransferProperties**

If you select **Use Scope Address** the file transfer processor uses the Scope Address entered in the Network Integrity UI.

If **Use Scope Address** is not selected, a String input parameter is required for the address. The address value must be in the form of *host*/*path*. Where *host* is either a host name, IPv4 or IPv6 address, and *path* is the directory path to where the files are located. For example, **192.168.1.1/tmp/test**. If the file transfer processor is only retrieving files from the local file system, do not specify the *host*. For example, **/tmp/test**.

Note: Only discovery actions have the Use Scope Address option because only discovery actions contain a scope address. For assimilation and import actions, there is no scope address, so the address must come from a context parameter.

Table 2–2 summarizes the required input parameters based on the configuration.

Table 2–2 Input Parameters Based on Configuration

Parameter Source	Use Scope Address	Required Input Context Parameters
Scan parameter group	Checked	None
Scan parameter group	Unchecked	String
Context Parameter	Checked	oracle.communications.sce.integrity.sdk.fileTransferCollector.FileTransferProperties
Context Parameter	Unchecked	oracle.communications.sce.integrity.sdk.fileTransferCollector.FileTransferProperties String

Setting File Transfer Properties

Context parameters of type **oracle.communications.sce.integrity.sdk.fileTransferCollector.FileTransferProperties** can be set in a predecessor action processor so that some or all of the properties can be defined in the action. That is, the action can be configured to specify all the file transfer properties, instead of prompting the user to enter values.

The File Transfer Property Initializer processor creates and populates a FileTransferProperties object. The object can then be used by the file transfer processor to transfer the file.

Example 2–1 shows the invoke method implementation of the file transfer property initializer. The code demonstrates how to programmatically set file transfer properties. In this example, the values are static, but the values can come from scan parameter groups defined on the action, or they can come from the result of an external system API call.

Example 2–1 Invoke Method Implementation

```
@Override
public FileTransferPropertyInitializerProcessorResponse invoke(
              DiscoveryProcessorContext context,
              FileTransferPropertyInitializerProcessorRequest request) throws
ProcessorException {
       FileTransferProperties ftProperties = new FileTransferProperties();

       ftProperties.setFilePattern("*.txt");
       ftProperties.setUser("someUser");
       ftProperties.setPassword("myPassword");
       ftProperties.setPort(21);
       ftProperties.setSessionTimeOut(120);
       ftProperties.setFileTransferType(FileTransferTypeT.FTP);
       ftProperties.setSrcFileManagement(SrcFileManagementTypeT.DELETE);

       return new FileTransferPropertyInitializerProcessorResponse(ftProperties);
}
```

FTP and SFTP Limitations

Network Integrity can fail to complete a file transfer using FTP or SFTP, and when the **Source File Management** is set to Rename, if the scope location already contains a file with the processed name. This problem is an inherent problem with FTP and SFTP. Ensure that the scope location does not contain a renamed file.

For example, to transfer and rename the file **sample.txt** from a remote location, ensure that the location does not also contain a file named **sample.txtProcessed**.

3 The File Parser Processor

This chapter provides information about the Oracle Communications Network Integrity file parser processor.

About the File Parser Processor

The file parser processor is code-generated and can parse XML and structured ASCII files into Java representations, which can then be processed by other processors. For information about the ASCII reference cartridge, see "The ASCII Reference Cartridge". For information about the XML reference cartridge, see "The XML Reference Cartridge".

File parser processors can be used within discovery, import, and assimilation actions.

The file parser processor receives a collection of files as its input. The input context parameter is of type **java.util.Collection<java.io.File>**. The input context parameter typically comes from a file transfer processor, but it can come from any processor that outputs the proper type. A processor with a custom implementation, which outputs a **java.util.Collection<java.io.File>** context parameter, can supply input to the file transfer processor as well.

The file parser processor returns an iterator output context parameter. For XML files, the iterator is of type DocumentWrappers. For ASCII files, the iterator is of type RecordWrappers. The iterator is typically used as an input parameter to a For Each processor. The For Each processor returns individual document or record wrappers on each iteration of the For Each loop.

Note: The names of the DocumentWrapper and RecordWrapper classes are derived from the name of the processor. For example, if the name of a processor that is used to parse XML documents is "XML File Parser," the name of the DocumentWrapper class is "XMLFileParserDocument." If the name of a processor that is used to parse ASCII files is "ASCII File Parser," the name of the RecordWrapper class is "ASCIIFileParserWrapper."

The file parser processor uses the iterator pattern to help reduce resource usage. It ensures that only one file is open at a time, and it also helps to reduce memory usage. When parsing ASCII files, only a single record is loaded into memory at a time. When parsing XML files, only a single document is loaded into memory at a time. However, for large XML files, the memory use might still be significant.

Note: For very large XML files, consider using a processor with a custom implementation that uses a SAX-style parser.

If a processor throws an exception while using the iterator, the exception is caught by the action controller class. The action controller class calls the close method with a wasError parameter value of *true*, which causes the file parser processor to rename the current file to the same name but with a ".error" extension. This allows the file to be analyzed. A log file is also produced if this event occurs.

About the XML API

When configured to parse XML files, one of the main configuration parameters of the file parser processor is the name of an XML schema file that describes the documents to be parsed. From the schema file, the file parser processor generates an API to allow the interrogation and manipulation of the data in the XML files. The XML API consists of:

- A document wrapper class, which wraps an XMLBeans document and provides:
  - A method for checking if the document is valid. When a document is parsed, it is automatically validated against the schema. The isValid method determines if the document is a valid document.
  - If the document is not valid, it might have failed to be parsed. This usually means that the XML is not well formed. The method getParseException retrieves the exception that was thrown by the XMLBeans parser, which can sometimes be useful in diagnosing the reason for the parsing failure.
  - Schemas that have multiple top-level elements defined support different document types. The getDocumentType method determines the type of the document. The method returns a DocumentType enumeration value.
  - Methods for getting the wrapped XMLBeans document. There is a getter method for each document type, supported by the schema. If the user calls the wrong document getter method, null is returned.
  - The getFile method returns the file associated with the XML wrapper document.
  - A constructor that takes a *java.io.File*. The constructor is used by the iterator. It is not normally used by clients of the document wrapper class.
- The XMLBeans classes are the second and main part of the XML API. The schema is automatically compiled into XMLBeans. The XMLBeans provide the remainder of the XML API. XMLBeans is an open source technology. Documentation on XMLBeans and its APIs can be found at:

  `http://xmlbeans.apache.org/`

Example 3–1 shows sample code that demonstrates the use of the XML API generated by the file parser. This code is located in the modeler processor implementation class, as seen in the code below. This code is supplied by the cartridge developer.

Example 3–1 File Parser-generated XML API

```
package com.oracle.integrity.xmlexamplecartridge.discoveryprocessors.xmlmodeller;

import java.util.logging.Level;
import java.util.logging.Logger;
```

```
import
com.oracle.integrity.xmlexamplecartridge.fileparserprocessors.examplexmlparserproc
essor.ExampleXMLParserProcessorDocument;

import oracle.communications.integrity.scanCartridges.sdk.ProcessorException;
import
oracle.communications.integrity.scanCartridges.sdk.context.DiscoveryProcessorConte
xt;

public class XMLModellerProcessorImpl implements XMLModellerProcessorInterface {
    private static Logger logger = Logger
            .getLogger(XMLModellerProcessorImpl.class.getName());

    @Override
    public void invoke(DiscoveryProcessorContext context,
            XMLModellerProcessorRequest request) throws ProcessorException {
        logger.log(Level.FINE, "Entering XMLModellerProcessorImpl");

        ExampleXMLParserProcessorDocument documentWrapper = request
                .getXmlDocument();
        if (!documentWrapper.isValid()) {
            logger.log(Level.WARNING, "Document for file '"
                    + documentWrapper.getFile()
                    + "' is invalid. Parse exception: "
                    + documentWrapper.getParseException());
        } else if (!(documentWrapper.getDocumentType() ==
                     ExampleXMLParserProcessorDocument.DocumentType.
BulkCmConfigDataFileDocument)) {
            logger.log(Level.WARNING, "Document for file '"
                    + documentWrapper.getFile()
                    + "' is invalid. Parse exception: "
                    + documentWrapper.getParseException());
        } else {
            // Get the XMLBeans document class
            BulkCmConfigDataFileDocument bulkCmConfigDataFileDocument =
documentWrapper
                    .getBulkCmConfigDataFileDocument();
            /*
             * Additional code, not shown here, would use the XMLBeans API to
             * access the information in the document.
             */
        }
        logger.log(Level.FINE, "Leaving XMLModellerProcessorImpl");
     }
}
```

About ASCII Record API

When configured for structured ASCII files, the cartridge developer supplies the rules for parsing the ASCII file. Included are the rules for parsing the header, body, and trailer records, and their fields. The header and trailer rules are optional, because, in some cases, they might not be required. From the rules, the file parser processor generates a Java API, which allows easy access to the information in the ASCII files. The RecordWrapper class is that API.

The RecordWrapper class provides the following:

- The getRecordType method returns the type of the wrapped record: Body, Header, and Trailer, for valid records, and Unknown for a record that fails to parse correctly. If a header record is not configured, the header value is not included in the RecordWrapper API. The same is true for trailer records.

- The methods getBodyRecord, getHeaderRecord, and getTrailerRecord return the wrapped BodyRecord, HeaderRecord, and TrailerRecoder classes. The BodyRecord, HeaderRecord, and TrailerRecord classes provide getter methods for retrieving each of the included fields of the record. If a record is configured to be ignored, its corresponding getter method is not available in the RecordWrapper API. Also, if the wrong get record method is called, the method returns null. (For example, if the RecordWrapper wraps a HeaderRecord, the getBodyRecord returns null.)
- The method getFile returns the file associated with the record.
- The method getRecordPosition returns, the character offset of the record within its file.

The sample code in Example 3–2 demonstrates the use of the ASCII API generated by the file parser processor. This code is located in the modeler processor implementation class. This code is supplied by the cartridge developer.

Example 3–2 File Parser-generated ASCII API

```
package com.oracle.integrity.asciicarparser.discoveryprocessors.asciimodeller;

import java.util.logging.Level;
import java.util.logging.Logger;

import
com.oracle.integrity.asciicarparser.fileparserprocessors.parseasciicars.ParseASCII
CarsWrapper;
import
com.oracle.integrity.asciicarparser.fileparserprocessors.parseasciicars.ParseASCII
CarsWrapper.BodyRecord;
import
com.oracle.integrity.asciicarparser.fileparserprocessors.parseasciicars.ParseASCII
CarsWrapper.HeaderRecord;
import
com.oracle.integrity.asciicarparser.fileparserprocessors.parseasciicars.ParseASCII
CarsWrapper.TrailerRecord;

import oracle.communications.integrity.scanCartridges.sdk.ProcessorException;
import
oracle.communications.integrity.scanCartridges.sdk.context.DiscoveryProcessorConte
xt;

public class ASCIIModellerProcessorImpl implements
        ASCIIModellerProcessorInterface {
    private static Logger logger = Logger
            .getLogger(ASCIIModellerProcessorImpl.class.getName());

    @Override
    public void invoke(DiscoveryProcessorContext context,
            ASCIIModellerProcessorRequest request) throws ProcessorException {
        logger.log(Level.FINE, "Entering ASCIIModellerProcessorImpl");

        ParseASCIICarsWrapper carWrapper = request.getCarRecord();

        if (carWrapper.getRecordType() == ParseASCIICarsWrapper.RecordType.Header)
{
            // For this example, our columns are "Make", "Model" and "Year".
            // Here we are verify that the header of the file is correct.
            if ((!headerRecord.getMakeColumn().equals("Make"))
                    || (!headerRecord.getModelColumn().equals("Model"))
                    || (!headerRecord.getYearColumn().equals("Year"))) {
                throw new ProcessorException("Error in header columns");
            }
        } else if (carWrapper.getRecordType() ==
ParseASCIICarsWrapper.RecordType.Body) {
            BodyRecord bodyRecord = carWrapper.getBodyRecord();

            logger.log(Level.FINE, "Car body record: " + bodyRecord.getMake()
                     + " " + bodyRecord.getModel() + " " + bodyRecord.getYear());
         } else if (carWrapper.getRecordType() ==
ParseASCIICarsWrapper.RecordType.Trailer) {
            TrailerRecord trailerRecord = carWrapper.getTrailerRecord();

            // For this example, the trailer record configuration has a single
            // field defined called "All" with Aggregate Extra Fields option
            // selected. This returns the full record as a single field.
            logger.log(Level.FINE, "Car trailer record: "
                    + trailerRecord.getAll());
        } else {
            throw new ProcessorException("Error parsing: "
                    + carWrapper.getFile());
        }
        logger.log(Level.FINE, "Entering ASCIIModellerProcessorImpl");
    }
}
```

ASCII Parsing Examples

The ASCII parsing examples in this section show how the files might be configured and how the information is then displayed in the file parser processor user interface.

Example: CSV File with Header, Body, and Trailer Records

Example 3–3 shows a CSV file with header, body, and trailer records.

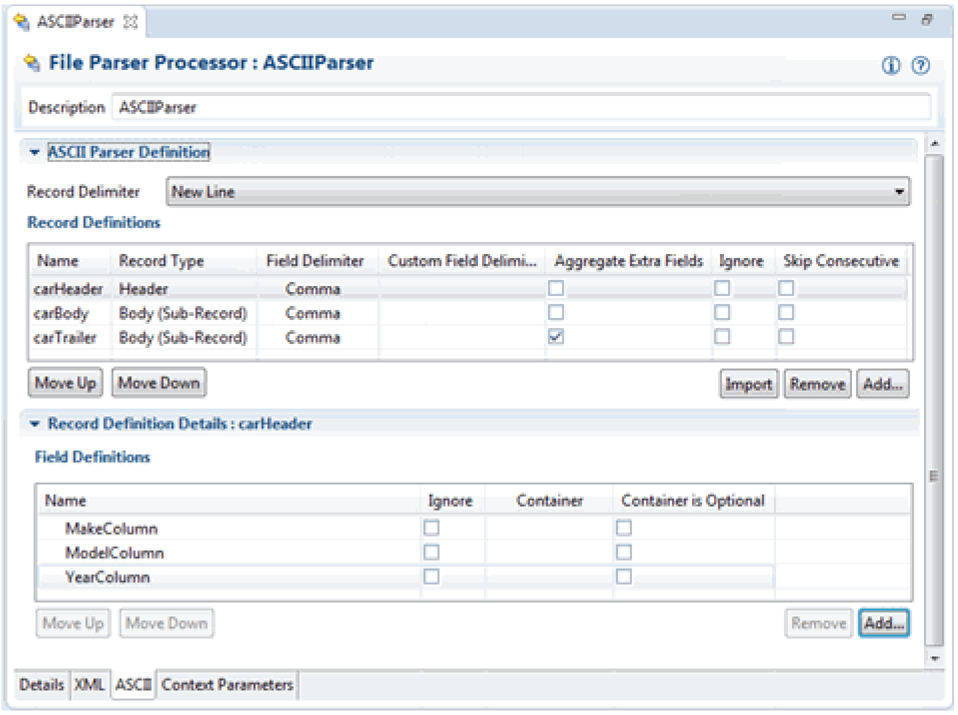
Example 3–3 CSV File with Header, Body, and Trailer Records

```
Make,Model,Year
Lamborghini,Murcielago,2003
Lamborghini,Gallardo,2007
Lamborghini,LP 640,2007
===========================
```

Figure 3–1 shows how the header record definition might be configured. With this definition, the API would include getMakeColumn, getModelColumn, and getYearColumn methods on the HeaderRecord, which could be used to validate that the correct type of file is being read.

Note: Using the Aggregate Extra Fields option causes the full line of data to be returned by the getAll method.

Figure 3–1 Car Header Record Definition



ASCIIParser

File Parser Processor : ASCIIParser

Description: ASCIIParser

ASCII Parser Definition

Record Delimiter: New Line

Record Definitions

Name	Record Type	Field Delimiter	Custom Field Delimi...	Aggregate Extra Fields	Ignore	Skip Consecutive
carHeader	Header	Comma		☐	☐	☐
carBody	Body (Sub-Record)	Comma		☐	☐	☐
carTrailer	Body (Sub-Record)	Comma		☑	☐	☐

Move Up | Move Down | Import | Remove | Add...

Record Definition Details : carHeader

Field Definitions

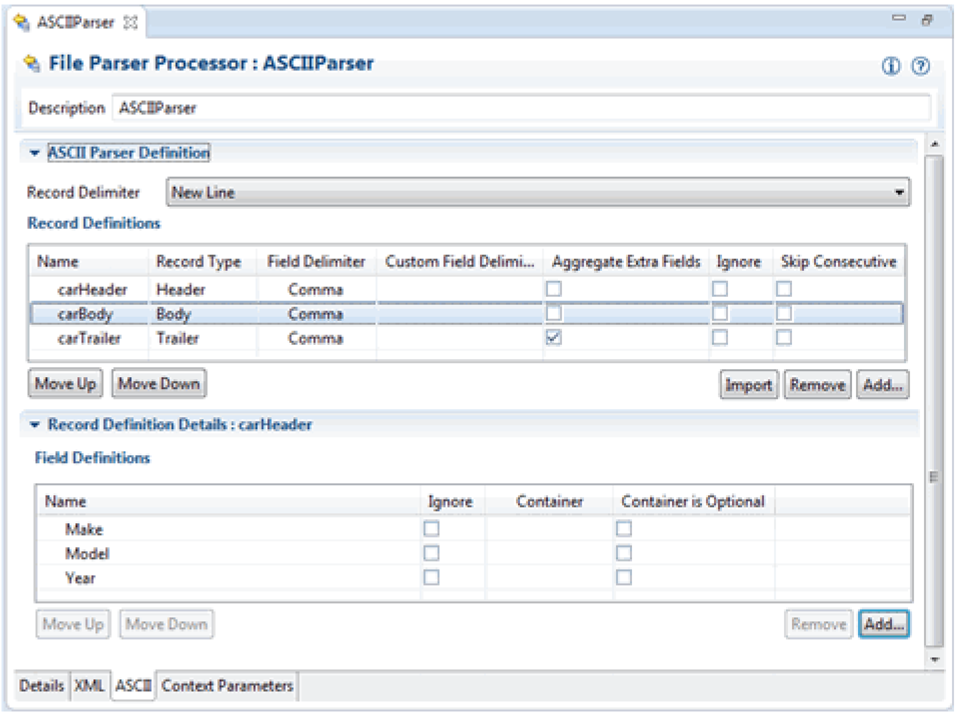
Name	Ignore	Container	Container is Optional
MakeColumn	☐		☐
ModelColumn	☐		☐
YearColumn	☐		☐

Move Up | Move Down | Remove | Add...

Details | XML | ASCII | Context Parameters

Figure 3–2 shows how the body record definition might be configured. With this definition, the API would include getMake, getModel, and getYear methods on the BodyRecord, which could be used to field values from the body record.

Figure 3–2 Car Body Record Definition



ASCIIParser

File Parser Processor : ASCIIParser

Description: ASCIIParser

ASCII Parser Definition

Record Delimiter: New Line

Record Definitions

Name	Record Type	Field Delimiter	Custom Field Delimi...	Aggregate Extra Fields	Ignore	Skip Consecutive
carHeader	Header	Comma		☐	☐	☐
carBody	Body	Comma		☐	☐	☐
carTrailer	Trailer	Comma		☑	☐	☐

Move Up | Move Down | Import | Remove | Add...

Record Definition Details : carHeader

Field Definitions

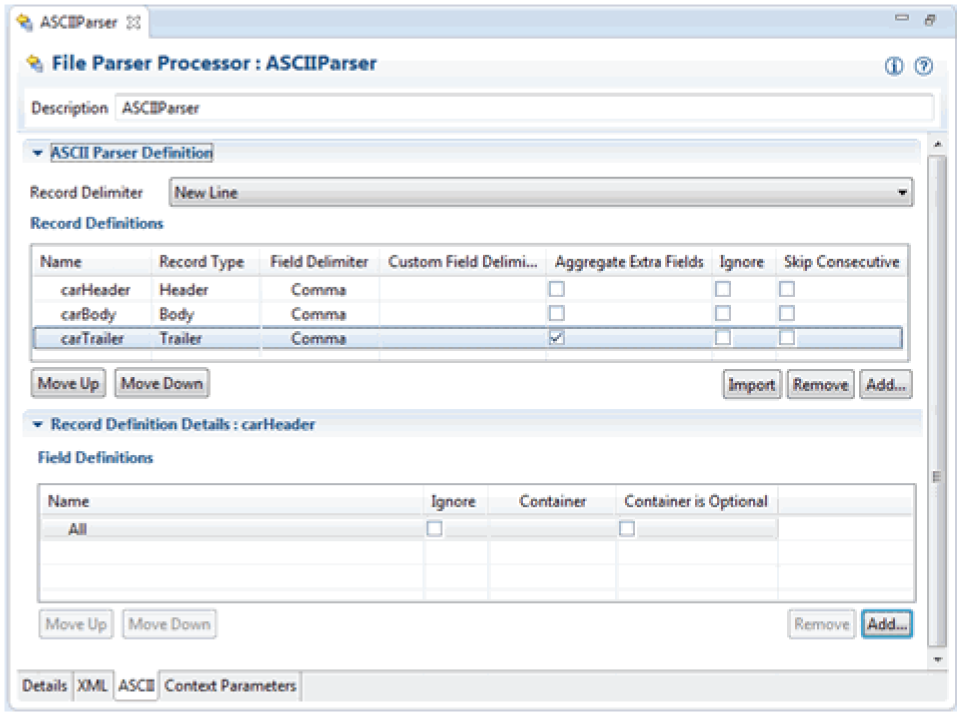
Name	Ignore	Container	Container is Optional
Make	☐		☐
Model	☐		☐
Year	☐		☐

Move Up | Move Down | Remove | Add...

Details | XML | ASCII | Context Parameters

Figure 3–3 shows how the trailer record might be configured. With this definition, the API would include a getAll method on the TrailerRecord.

Figure 3–3 Car Trailer Record Definition



File Parser Processor : ASCIIParser

Description: ASCIIParser

ASCII Parser Definition

Record Delimiter: New Line

Record Definitions

Name	Record Type	Field Delimiter	Custom Field Delimi...	Aggregate Extra Fields	Ignore	Skip Consecutive
carHeader	Header	Comma		☐	☐	☐
carBody	Body	Comma		☐	☐	☐
carTrailer	Trailer	Comma		✓	☐	☐

Record Definition Details : carHeader

Field Definitions

Name	Ignore	Container	Container is Optional
All	☐		☐

Example: ASCII File with Multi-line Records

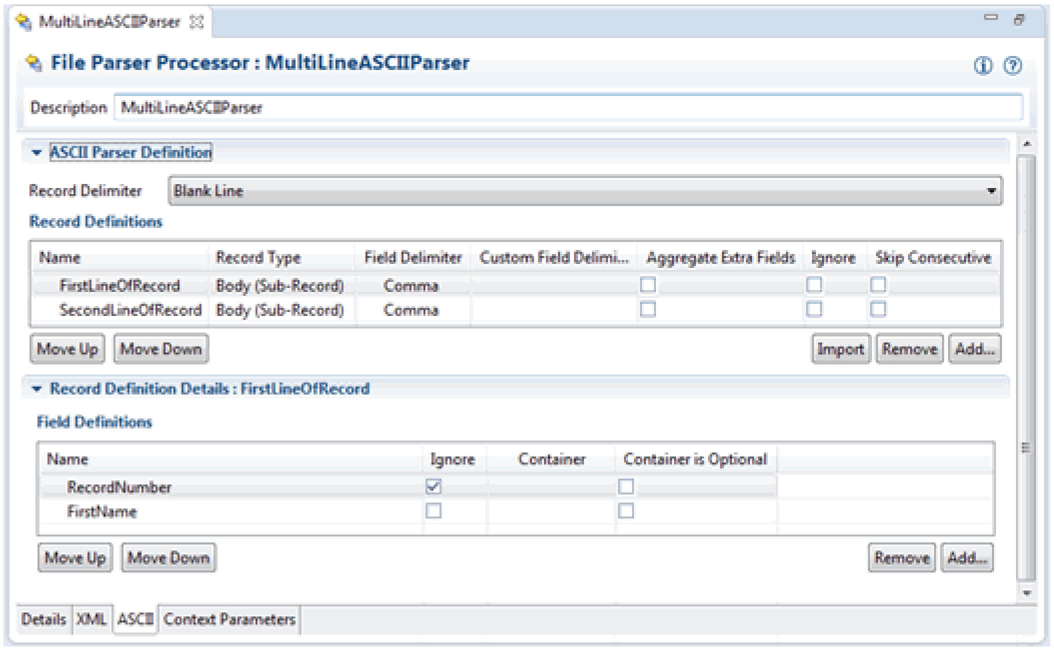
Example 3–4 shows an ASCII file that has multi-line records.

Example 3–4 ASCII File with Multi-line Records

```
1, John
Doe,8000

2,Dave
Smith,8001

3,Jim
Yong,8002

4,Kate
May,8003
```

Figure 3–4 and Figure 3–5 show how the body record definitions can be configured for this record structure. In this example, the record delimiter is a blank line. The Ignore option is set on the RecordNumber field. With this definition, the API would include getFirstName, getLastName, and getID methods on the BodyRecord.

Figure 3–4 Multi-line First Line Body Record Definition



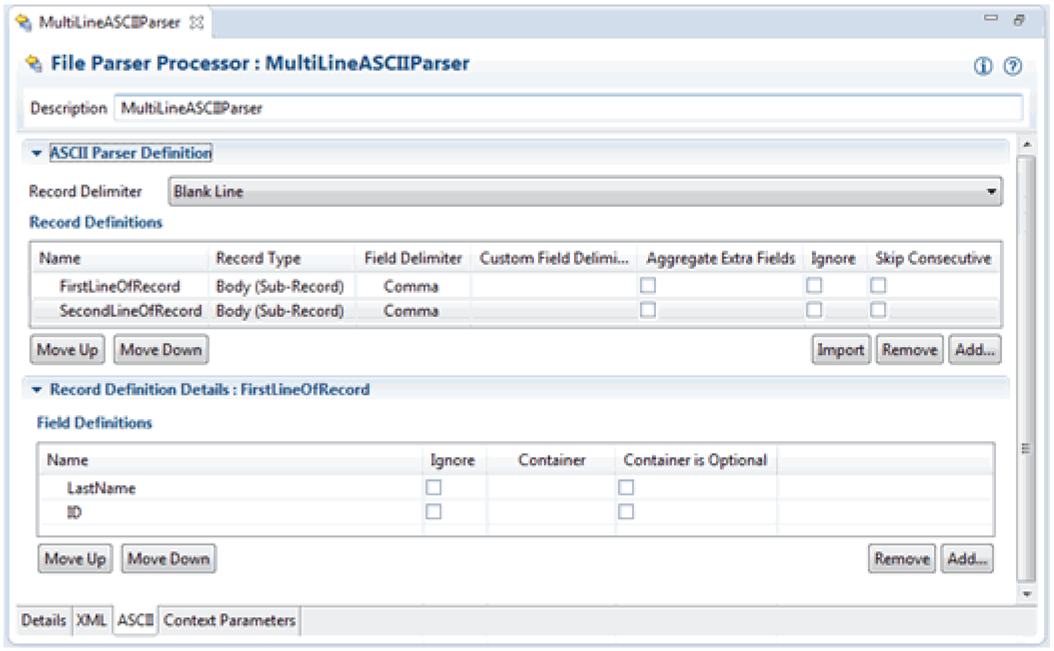
Description: MultiLineASCIIParser

Record Delimiter: Blank Line

Name	Record Type	Field Delimiter	Custom Field Delimi...	Aggregate Extra Fields	Ignore	Skip Consecutive
FirstLineOfRecord	Body (Sub-Record)	Comma		☐	☐	☐
SecondLineOfRecord	Body (Sub-Record)	Comma		☐	☐	☐

Name	Ignore	Container	Container is Optional
RecordNumber	☑		☐
FirstName	☐		☐

Figure 3–5 Multi-line Second Line Body Record Definition



Description: MultiLineASCIIParser

Record Delimiter: Blank Line

Name	Record Type	Field Delimiter	Custom Field Delimi...	Aggregate Extra Fields	Ignore	Skip Consecutive
FirstLineOfRecord	Body (Sub-Record)	Comma		☐	☐	☐
SecondLineOfRecord	Body (Sub-Record)	Comma		☐	☐	☐

Name	Ignore	Container	Container is Optional
LastName	☐		☐
ID	☐		☐

4

The ASCII Reference Cartridge

This chapter describes the functionality and design of the Oracle Communications Network Integrity ASCII Reference cartridge.

The ASCII Reference cartridge uses ASCII file processing technology. The cartridge uses an Alcatel 1359IOO Remote Inventory Data as its example. This guide assumes that you are familiar with remote inventory data handoff in Alcatel 1359IOO Information Content Description.

About Alcatel 1359IOO Remote Inventory Data Handoff

Alcatel 1359IOO uses a TCP/IP connection to exchange a set of messages between EOS and a generic IOO agent.

For a discovery action, the focus is on the remote inventory data handoff. Remote Inventory Data Handoff (RIDH) allows the export of the main identification data relevant to hardware products (physical boards) installed into the network elements (NEs) managed by the EML. The identification data is stored in permanent memory devices (EEPROM) included in each product part list.

This IOO application enables the EOS to retrieve all the remote inventory data stored in the 1353NM.

RI_DATA_NOTIFICATION primitives provide the EOS with the list of hardware components currently installed in the Network Elements (NE), and notify a change if the remote inventory file is updated on the 1353NM. Each RI_DATA_NOTIFICATION primitive refers to a single board.

RI_DATA_NOTIF primitives are sent after all the RI_DATA_NOTIFICATIONs referring to the boards installed in a single network element, to give the EOS an End Of File indicator (the inventory of the NEX is completely uploaded).

This reference implementation deals only with RI_DATA_NOTIF, because the Alcatel 1359 IOO Information Content Description document does not show RI_DATA_NOTIFICATION in the sample Remote Inventory Data. The Alcatel 1359 IOO Information Content Description document only shows RI_DATA_NOTIF. RI_DATA_NOTIFICATION does not appear in the RI Data Handoff document.

RI_DATA_NOTIF primitives have different parameters for Q3 and QB3* NEs. In this reference implementation, QB3* NE is used as example.

The CSV format of RI_DATA_NOTIF is as follows (QB3* NE):

```
RI_DATA_NOTIF
[ <riDataList or riDataUnsol> |
<neName attribute value> |
<neLocationName attribute value> |
<protocolType attribute value> |
<blockNumber attribute value> |
<blockLabel attribute value> | (unique to SND NE's)
<alcatelCompany attribute value> |
<unitType attribute value> |
<unitPartNumber attribute value> |
<softwarePartNumber attribute value> |
<cleiCode attribute value> |
<manufacPlant attribute value> |
<serialNumber attribute value> |
<manufacDate attribute value> |
<operatorInvData attribute value> ]
```

The CSV format for the last notification received for each remote inventory file is as follows:

```
RI_DATA_NOTIF
[ <riDataList or riDataUnsol > |
<neName attribute value> |
<protocolType attribute value> |
{ <uploadTime attribute value> } |
< numberOfCards attribute value> ]
```

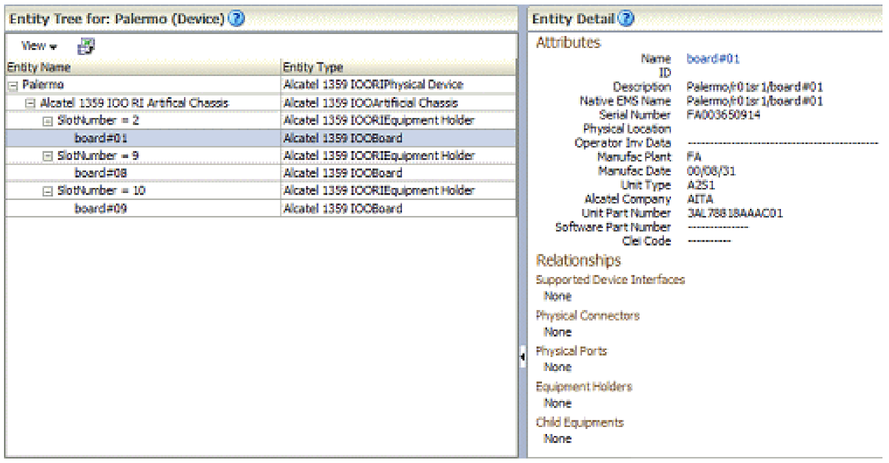
The ASCII Reference cartridge is designed to be used on a standalone basis to display the physical device hierarchy in Network Integrity. The ASCII Reference cartridge provides no integration with other products, but can be extended.

Modeling a Physical Device Hierarchy

The samples in this section do not directly deal with the TCP/IP protocol, as described in Alcatel 1359IOO document, to get the Remote Inventory Data. Instead, it is assumed that the remote inventory data is retrieved using the IOO protocol and stored as CSV files by an external process. The samples use the ASCII Reference Cartridge to retrieve the IOO CSV file, parse it, and model it into Oracle Communications Information Model. Each RI_DATA_NOTIF record contains one board for an NE. All the boards belonging to one NE must be aggregated as a list of child equipment of the NE (NE is modeled as Physical Device).

Figure 4–1 shows a sample discovered physical device hierarchy. This hierarchy is displayed in the Network Integrity user interface, in the Scan Result Detail page.

Figure 4–1 Sample Discovered Physical Device Hierarchy



Entity Name	Entity Type
Palermo	Alcatel 1359 IOORIPhysical Device
Alcatel 1359 IOO RI Artifical Chassis	Alcatel 1359 IOOArtificial Chassis
SlotNumber = 2	Alcatel 1359 IOORIEquipment Holder
board#01	Alcatel 1359 IOOBoard
SlotNumber = 9	Alcatel 1359 IOORIEquipment Holder
board#08	Alcatel 1359 IOOBoard
SlotNumber = 10	Alcatel 1359 IOORIEquipment Holder
board#09	Alcatel 1359 IOOBoard

Attributes

Name	board#01
ID	
Description	Palermo/r01sr1/board#01
Native EMS Name	Palermo/r01sr1/board#01
Serial Number	FA003650914
Physical Location	
Operator Inv Data	--
Manufac Plant	FA
Manufac Date	00/08/31
Unit Type	A2S1
Alcatel Company	AITA
Unit Part Number	3AL78818AAAC01
Software Part Number	--------------
Clei Code	----------

Relationships

Supported Device Interfaces
None

Physical Connectors
None

Physical Ports
None

Equipment Holders
None

Child Equipments
None

Cartridge Dependencies

This section provides information about dependencies that the ASCII Reference cartridge has on other entities.

Run-Time Dependencies

For the ASCII Reference cartridge to work at run time, you must deploy the Address_Handlers cartridge to Network Integrity.

Design-Time Dependencies

The ASCII Reference cartridge has the following dependencies:

- Address_Handlers
- NetworkIntegritySDK
- ora_uim_model

Opening ASCII Reference Cartridge Files

This section provides information about downloading and opening the ASCII Reference cartridge files in Design Studio. After you open the files, you can review and extend them.

You can download a ZIP file that contains the individual Design Studio files. You can open these files in Design Studio to review and extend the cartridge ZIP files.

Opening Files in Design Studio

To review and extend the ASCII Reference cartridge, you must first download the Oracle Communications Network Integrity File Transfer and Parsing software from the Oracle software delivery website:

`https://edelivery.oracle.com`

The software contains the ASCII Reference cartridge ZIP file, which has the following structure:

- **\Network_Integrity_Cartridge_Projects\ASCII_Reference_Cartridge**

For information about opening files in Design Studio, see the Design Studio Help and *Network Integrity Developer's Guide*.

Compiling and Deploying the Cartridge

This section provides information about compiling and deploying the ASCII Reference cartridge.

To compile and deploy the ASCII Reference cartridge:

1. Import projects into Design Studio for Network Integrity.
2. Clean and build the cartridge.
3. Deploy the cartridge.

For more information about deploying and undeploying, see *Network Integrity Developer's Guide*.

About the Cartridge Components

The ASCII Reference Cartridge contains the following actions:

- Discover Alcatel 1359 IOO RI File

Discover Alcatel 1359 IOO RI File

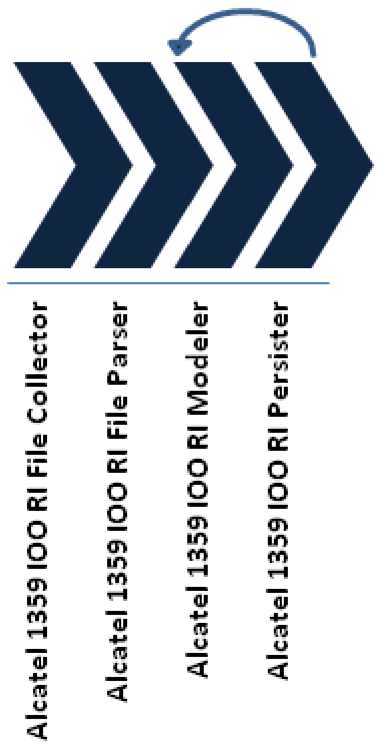
The Discover Alcatel 1359IOO RI action reads one or more Alcatel 1359IOO RI CSV file instances in a directory, and from it provides hierarchical physical device model instances.

The Discover Alcatel 1359 IOO RI File action contains the following processors run in the following order:

1. Alcatel 1359IOO RI File Collector
2. Alcatel 1359IOO RI File Parser
3. Alcatel 1359IOO RI Modeler
4. Alcatel 1359IOO RI Persister

Figure 4–2 illustrates the processor workflow of the Discover Alcatel 1359 IOO RI File action.

Figure 4–2 Discover Alcatel 1359IOO RI File Action Processor Workflow


Alcatel 1359 IOO RI File Collector
Alcatel 1359 IOO RI File Parser
Alcatel 1359 IOO RI Modeler
Alcatel 1359 IOO RI Persister

Alcatel 1359IOO RI File Collector

The Alcatel 1359IOO RI File Collector processor is used to retrieve Alcatel 1359IOO RI CSV files, which are then made available to the next processor in the chain.

Note: This processor is automatically generated from Design Studio input data.

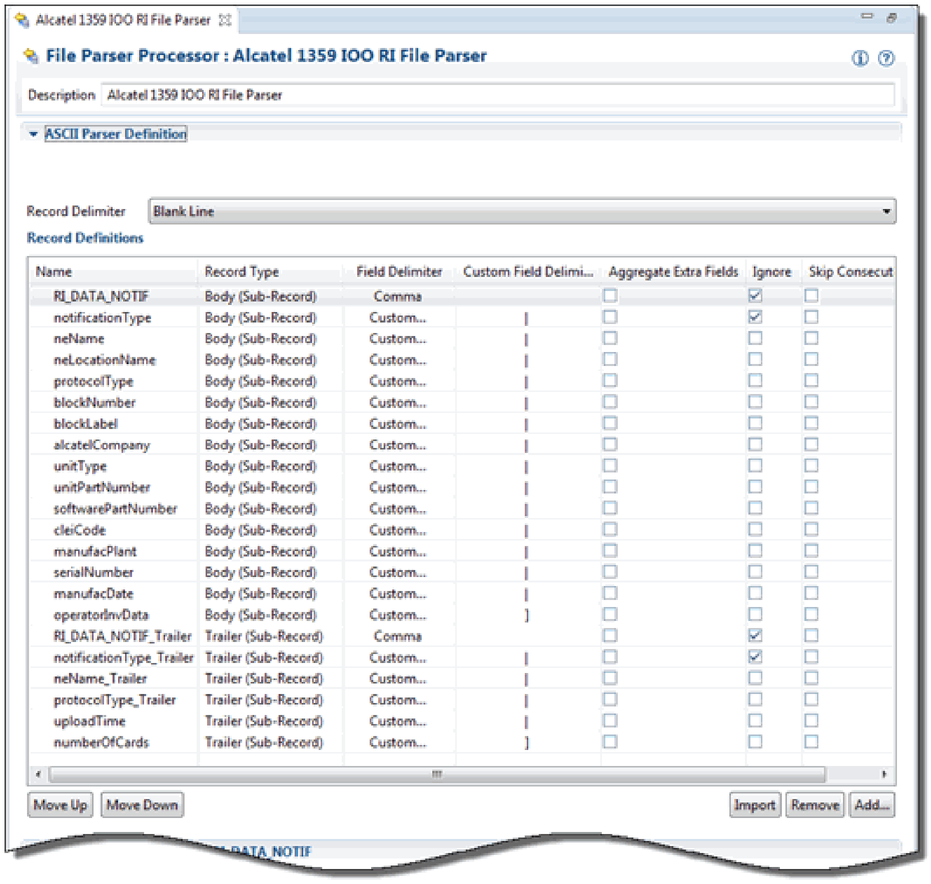
Alcatel 1359IOO RI File Parser

The Alcatel 1359IOO RI File Parser processor is used to read the RI CSV files, parse the CSV file to get a list of RI records, and make them available to the next processor.

Each Alcatel IOO RI data record consists of multiple lines (refer to the sample IOO RI data in "About Collected Data"). To parse the multiple lines of record, each line must be defined as a sub-record. The last record in the IOO RI CSV file is the end of data record, which is configured as the trailer record. Like the data record, the trailer record, is a multi-line record, so each line is configured as a sub-record of the trailer record. See the Alcatel 1359IOO RI File Parser processor in Design Studio for information about how to configure the ASCII parsing rules for Alcatel 1359IOO RI CSV file.

The following diagram shows the ASCII parsing rules configuration tab of the Alcatel 1359IOO RI File Parser in Design Studio.

Figure 4–3 ASCII Parser in Design Studio


Alcatel 1359 IOO RI File Parser
File Parser Processor : Alcatel 1359 IOO RI File Parser
ASCII Parser Definition
Move Up
Move Down
Import
Remove
Add...
RI_DATA_NOTIF

Description: Alcatel 1359 IOO RI File Parser

Record Delimiter: Blank Line

Record Definitions

Name	Record Type	Field Delimiter	Custom Field Delimi...	Aggregate Extra Fields	Ignore	Skip Consecut
RI_DATA_NOTIF	Body (Sub-Record)	Comma		☐	☑	☐
notificationType	Body (Sub-Record)	Custom...	|	☐	☑	☐
neName	Body (Sub-Record)	Custom...	|	☐	☐	☐
neLocationName	Body (Sub-Record)	Custom...	|	☐	☐	☐
protocolType	Body (Sub-Record)	Custom...	|	☐	☐	☐
blockNumber	Body (Sub-Record)	Custom...	|	☐	☐	☐
blockLabel	Body (Sub-Record)	Custom...	|	☐	☐	☐
alcatelCompany	Body (Sub-Record)	Custom...	|	☐	☐	☐
unitType	Body (Sub-Record)	Custom...	|	☐	☐	☐
unitPartNumber	Body (Sub-Record)	Custom...	|	☐	☐	☐
softwarePartNumber	Body (Sub-Record)	Custom...	|	☐	☐	☐
cleiCode	Body (Sub-Record)	Custom...	|	☐	☐	☐
manufacPlant	Body (Sub-Record)	Custom...	|	☐	☐	☐
serialNumber	Body (Sub-Record)	Custom...	|	☐	☐	☐
manufacDate	Body (Sub-Record)	Custom...	|	☐	☐	☐
operatorInvData	Body (Sub-Record)	Custom...	]	☐	☐	☐
RI_DATA_NOTIF_Trailer	Trailer (Sub-Record)	Comma		☐	☑	☐
notificationType_Trailer	Trailer (Sub-Record)	Custom...	|	☐	☑	☐
neName_Trailer	Trailer (Sub-Record)	Custom...	|	☐	☐	☐
protocolType_Trailer	Trailer (Sub-Record)	Custom...	|	☐	☐	☐
uploadTime	Trailer (Sub-Record)	Custom...	|	☐	☐	☐
numberOfCards	Trailer (Sub-Record)	Custom...	]	☐	☐	☐

Note: This processor is automatically generated from Design Studio by configuring a set of proper ASCII parsing rules.

Alcatel 1359IOO RI Modeler

The Alcatel 1359IOO RI Modeler processor is used to model each individual RI record that is parsed by the Alcatel1359IOORIFileParser processor and aggregate all of them into a single physical device entity. This processor demonstrates how to do aggregation when modeling data in Network Integrity.

Alcatel 1359IOO RI Persister

The Alcatel 1359IOO RI Persister processor is used to persist the physical device tree to the Network Integrity database.

About Collected Data

This section shows a sample Alcatel 1359IOO RI CSV file that is provided to the Alcatel1359IOORIFileParser processor. This CSV file is generated by an external process, which uses the IOO protocol (TCP/IP connection) to get the RI records from EOS and save them to an ASCII file. This ASCII file is collected by the Alcatel1359IOORIFileCollector processor.

One ASCII file contains the information of all the boards for one NE. The ASCII file ends with the end of data record (the last record in the following sample ASCII file) to indicate that there are no more boards from that NE.

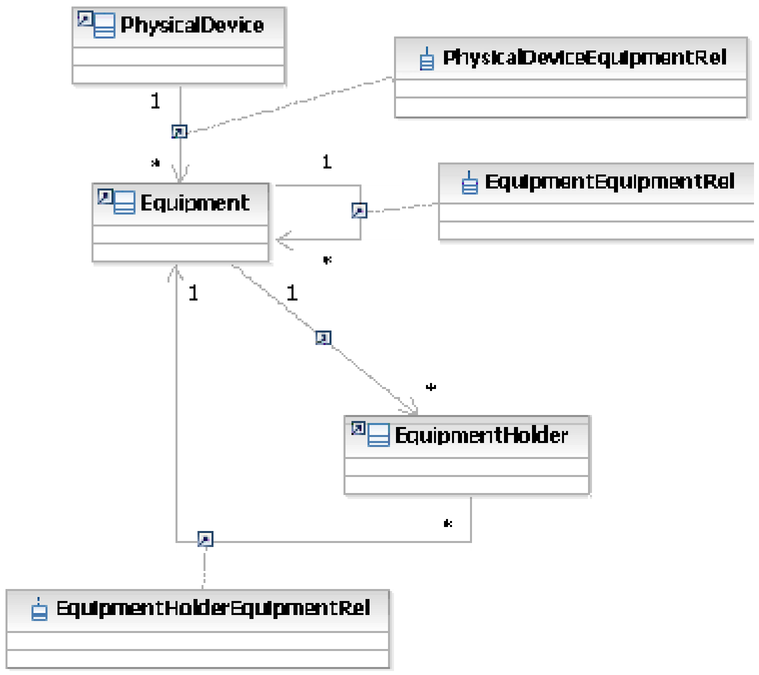
```
RI_DATA_NOTIF
[riDataList|
Palermo|
Zen|
QB3*|
1|
Palermo/r01sr1/board#01|
AITA|
A2S1|
3AL78818AAAC01|
--------------|
----------|
FA|
FA003650914|
00/08/31|
---------------------------------------------]

RI_DATA_NOTIF
[riDataList|
Palermo|
Zen|
QB3*|
8|
Palermo/r01sr1/board#08|
AITA|
PREA4ETH|
3AL79631AAAC03|
--------------|
----------|
FA|
FA024658237|
03/05/07|
---------------------------------------------]

RI_DATA_NOTIF
[riDataList|
Palermo|
Zen|
QB3*|
9|
Palermo/r01sr1/board#09|
AITA|
SYNTH1N|
3AL79090BAAA01|
--------------|
----------|
EZ|
EZ004150316|
00/10/06|
EXP RAM 32MB----------------------------------]

RI_DATA_NOTIF
[riDataList|
Palermo|
QB3*|
{2010/12/10 12:25:32}|
3]
```

About Cartridge Modeling

This section provides information about modeling the ASCII Reference cartridge.

Figure 4–4 shows a Unified Modeling Language (UML) diagram depicting the object relationship being rendered.

Figure 4–4 Information Model Entities UML Diagram



Hierarchy Mapping

The physical device object is established and seeded with data sourced by "neName" attribute inside the RI record.

The Equipment object (board) is established and seeded from "blockLabel" attribute. Artificial chassis are created to the NE so that slot (equipment holder) can be created under.

The EquipmentHolder object is established and seeded from "blockNumber" attribute and modeled as slot: Block Number = Slot Number - 1.

Oracle Communications Information Model Information

All entities shown in Figure 4–4 (for example, physical device, and equipment) are Information Model 1.0-compliant for static fields. The dynamic fields (sometimes referred to as characteristics) are application-specific.

Field Mapping

This section provides information about field mappings used in the cartridge.

- Text: Implies Text [255].
- static: The Information Model 1.0 defines this field to be static on the entity specification. The specification provides getters/setters for this field.
- dynamic: This is a dynamic field where the entity specification treats the field as a name/value pair. The specification does not provide getter/setters but generically has a get/setCharacteristics method holding a HashSet of entries.

Table 4–1 Physical Device Mappings

Physical Device	Information Model Support	RI Record Attribute	Field Type
Id	static	N/A	Text
Name	static	neName	Text
Description	static	N/A	Text
Specification	static	N/A	N/A, Programmatically set to Alcatel1359IOORIPhysicalDevice
neLocationName	static	neLocationName	Text
protocolType	dynamic	protocolType	Text
nativeEmsName	static	neName	Text
discoveredVendorName	dynamic	N/A	Text, hard-coded to be set to Alcatel
Serial Number	yes	N/A	Text
Physical Location	yes	N/A	Text

Table 4–2 Equipment Mappings

Equipment	Information Model Support	RI Record Attribute	Field Type
Id	static	N/A	Text
Name	static	blockLabel	Text, Extract the last part of blockLabel
Description	static	blockLabel	Text
Specification	static	N/A	N/A

Table 4–2 (Cont.) Equipment Mappings

Equipment	Information Model Support	RI Record Attribute	Field Type
alcatelCompany	dynamic	alcatelCompany	Text, Programmatically set to Alcatel1359IOORIPhysicalDevice
unitType	dynamic	unitType	Text
unitPartNumber	dynamic	unitPartNumber	Text
softwarePartNumber	dynamic	softwarePartNumber	Text
cleiCode	dynamic	cleiCode	Text
manufacPlant	dynamic	manufacPlant	Text
manufacDate	dynamic	manufacDate	Text
operatorInvData	dynamic	operatorInvData	Text
serialNumber	static	serialNumber	Text
nativeEmsName	static	blockLabel	Text
discoveredVendorName	dynamic	N/A	Text, hard-coded to be set to Alcatel
Physical Location	yes	N/A	Text

Table 4–3 EquipmentHolder Mappings

EquipmentHolder	Information Model Support	RI Record Attribute	Field Type
Id	static	N/A	Text
Name	static	blockNumber	Text, SlotNumber = blockNumber + 1
Specification	static	N/A	N/A, Programmatically set to Alcatel1359IOORIEquipmentHolder
nativeEmsName	static	blockNumber	Text, SlotNumber = BlockNumber +1
Description	yes	N/A	Text
Serial Number	yes	N/A	Text
Physical Location	yes	N/A	Text

Model Correction

This section provides Alcatel 1359IOO RI to Oracle Communications Information Model correction information.

About Model Correction Code

Model correction occurs when the Alcatel 1359IOO RI information received through discovery does not conform to Information Model and therefore cannot be persisted, as it is within Network Integrity. See "About Cartridge Modeling" for supported hierarchy.

The ASCII Reference Cartridge applies the model corrections as outlined below.

EquipmentHolder under physical device:

```
PhysicalDevice
   EquipmentHolder
```

The ASCII Reference Cartridge adds an equipment entity as follows:

```
PhysicalDevice
  Equipment-named Alcatel 1359IOO RI Artificial Chassis
   EquipmentHolder
```

Design Studio Construction

This section provides information about using Design Studio to construct the ASCII Reference cartridge.

The ASCII Reference cartridge contains the following specifications:

- Alcatel1359IOORIPhysicalDevice
- Alcatel1359IOOBoard
- Alcatel1359IOORIEquipentHolder
- Alcatel1359IOOArtificialChassis

Table 4–4 Discover Alcatel 1359IOO RI File Action Construction

Result Category	Address Handler	Scan Parameter Group Characteristics	Model	Processors
Device	FileTransfer AddressHandler	ftaFileTransferType ftaFilePattern ftaPort ftaUser ftaPassword ftaSessionTimeOut ftaSourceFileManagement ftaRenameSuffix	Alcatel 1326 IOO RI Model	Alcatel 1359IOO RI File Collector Alcatel 1359IOO RI File Parser Alcatel 1359IOO RI Modeler Alcatel 1359IOO RI Persister

In Figure 4–2 the first two chevrons indicate code-generated processors from Design Studio user input.

- Alcatel 1359IOO RI File Collector is an instance of the file transfer processor
- Alcatel 1359IOO RI File Parser is an instance of the file parser processor

Table 4–5 Action Conditions for Discover Alcatel 1359IOO RI File Action

Condition Name	Notes
checkPhysicalDevice	This condition returns a false result if the physical device from the Alcatel 1359IOO RI Modeler is null.

Table 4–6 Discover Alcatel 1359IOO RI File Action Processors

Processor Name	Variables
Alcatel 1359IOO RI File Collector	Input: N/A Output: ■ alcatel1359IOORIFileCollectorFileCollection. *java.util.Collecton* A collection of files found in the path specified in the scope field.
Alcatel 1359IOO RI File Parser	Input: alcatel1359IOORIFileCollectorFileCollection Output: ■ alcatel1359IOORIFileParserIterable An iterable to iterate over each discovered file.
Alcatel 1359IOO RI Modeler	Input: riFile Output: physicalDevice This processor can be extended to enhance an individual physical device tree. Any processor that uses its output parameter must check if this value is null before using it. The physical device is null if it does not contain information from all the boards.
Alcatel 1359IOO RI Persister	Input: N/A Output: N/A Context is persisted for performance

Design Studio Extension

This section provides information about Design Studio extensions to the ASCII Reference cartridge.

The source code to this cartridge is provided. You can change any part to customize this cartridge to fit your environment.

The Alcatel 1359IOO RI Modeler aggregates the information for all the boards before completely modeling a physical device. Before a physical device is completely modeled, this processor outputs a null physical device. The action has a condition applied on the Alcatel 1359IOO RI Persister, which checks the physical device to determine whether it is null or not. The persister does not get invoked if PhysicalDevice is null (which means it is not completely modeled at that time). A new processor that is to further modify PhysicalDevice must apply the same condition (checkPhysicalDevice) to make sure that the physical device is ready and not null.

For more information on extensibility, see *Network Integrity Developer's Guide*.

5 The XML Reference Cartridge

This chapter describes the functionality and design of the Oracle Communications Network Integrity XML Reference cartridge.

The XML Reference cartridge uses the XML File processing technology. The cartridge uses an Ericsson XML device file as its example.

The XML Reference cartridge is designed to be used on a standalone basis to display the physical device hierarchy in the Network Integrity UI. The XML Reference Cartridge provides no integration with other products but can be extended.

This section assumes that you are familiar with the following:

- Network Inventory Organizer (NIO) Export Interface (v 155 19-APR 901 219 Uen F 2006-06-01)
- 3GPP TS 32.615v630

  (`http://www.3gpp.org/ftp/specs/html-info/32615.htm`)
- 3GPP TS 32.625v660

  (`http://www.3gpp.org/ftp/specs/html-info/32625.htm`)
- 3GPP TS 32.695v600

  (`http://www.3gpp.org/ftp/specs/html-info/32695.htm`)

Modeling a Physical Device Hierarchy

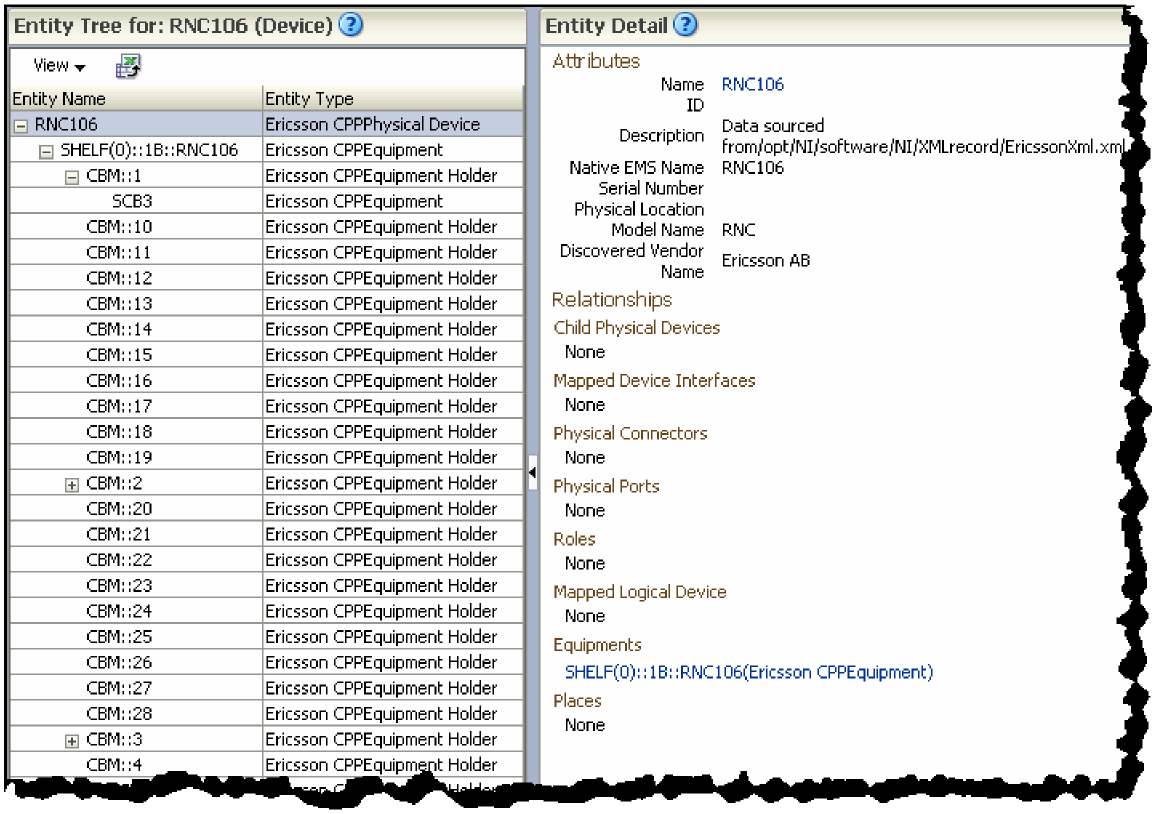
Using a CLI command, Ericsson devices can deliver XML device inventory to the local file system. These XML device files can be transferred to Network Integrity for processing.

See Ericsson SMO CLI, Software Management, Organizer Command Line Interface, User Guide, 2/1553-APR 901 007 for reference.

The cartridge reads the XML device file and produces a physical device hierarchy that represents the discovered device and includes a physical device instance, equipment, and equipment holders. (PhysicalPorts are not rendered in the XML file and so are not supported.)

Figure 5–1 shows a sample discovered physical device hierarchy. This hierarchy is displayed in the Network Integrity user interface, in the Scan Result Detail page.

Figure 5–1 Sample Discovered Physical Device Hierarchy



Entity Name	Entity Type
RNC106	Ericsson CPPPhysical Device
SHELF(0)::1B::RNC106	Ericsson CPPEquipment
CBM::1	Ericsson CPPEquipment Holder
SCB3	Ericsson CPPEquipment
CBM::10	Ericsson CPPEquipment Holder
CBM::11	Ericsson CPPEquipment Holder
CBM::12	Ericsson CPPEquipment Holder
CBM::13	Ericsson CPPEquipment Holder
CBM::14	Ericsson CPPEquipment Holder
CBM::15	Ericsson CPPEquipment Holder
CBM::16	Ericsson CPPEquipment Holder
CBM::17	Ericsson CPPEquipment Holder
CBM::18	Ericsson CPPEquipment Holder
CBM::19	Ericsson CPPEquipment Holder
CBM::2	Ericsson CPPEquipment Holder
CBM::20	Ericsson CPPEquipment Holder
CBM::21	Ericsson CPPEquipment Holder
CBM::22	Ericsson CPPEquipment Holder
CBM::23	Ericsson CPPEquipment Holder
CBM::24	Ericsson CPPEquipment Holder
CBM::25	Ericsson CPPEquipment Holder
CBM::26	Ericsson CPPEquipment Holder
CBM::27	Ericsson CPPEquipment Holder
CBM::28	Ericsson CPPEquipment Holder
CBM::3	Ericsson CPPEquipment Holder
CBM::4	Ericsson CPPEquipment Holder

Attributes

Name	RNC106
ID	
Description	Data sourced from/opt/NI/software/NI/XMLrecord/EricssonXml.xml
Native EMS Name	RNC106
Serial Number	
Physical Location	
Model Name	RNC
Discovered Vendor Name	Ericsson AB

Relationships

Child Physical Devices
None

Mapped Device Interfaces
None

Physical Connectors
None

Physical Ports
None

Roles
None

Mapped Logical Device
None

Equipments
SHELF(0)::1B::RNC106(Ericsson CPPEquipment)

Places
None

Cartridge Dependencies

This section provides information about dependencies that the XML Reference cartridge has on other entities.

Run Time Dependencies

For the XML Reference cartridge to work at run time, you must deploy the Address_Handlers cartridge to Network Integrity.

Design-Time Dependencies

The XML Reference cartridge has the following dependencies:

- Address_Handlers
- NetworkIntegritySDK
- ora_uim_model

Opening XML Reference Cartridge Files

This section provides information about downloading and opening the XML Reference Cartridge files in Design Studio. After you open the files, you can review and extend them.

You can download a ZIP file that contains the individual Design Studio files. You can open these files in Design Studio to review and extend the cartridge ZIP files.

Opening Files in Design Studio

To review and extend the XML Reference cartridge, download the Oracle Communications Network Integrity File Transfer and Parsing software from the Oracle software delivery website:

`https://edelivery.oracle.com`

The software contains the XML Reference cartridge ZIP file, which has the following structure:

- **\Network_Integrity_Cartridge_Projects\XML_Reference_Cartridge**

For information about opening files in Design Studio, see the Design Studio Help and *Network Integrity Developer's Guide*.

Compiling and Deploying the Cartridge

This section provides information about compiling and deploying the XML Reference Cartridge.

To compile and deploy the XML Reference Cartridge:

1. Import projects into Design Studio for Network Integrity.
2. Clean and build the cartridge.
3. Deploy the cartridge.

For more information about deploying and undeploying, see *Network Integrity Developer's Guide*.

About the Cartridge Components

The XML Reference cartridge contains the following actions:

- Discover Ericsson Xml

Discover Ericsson Xml

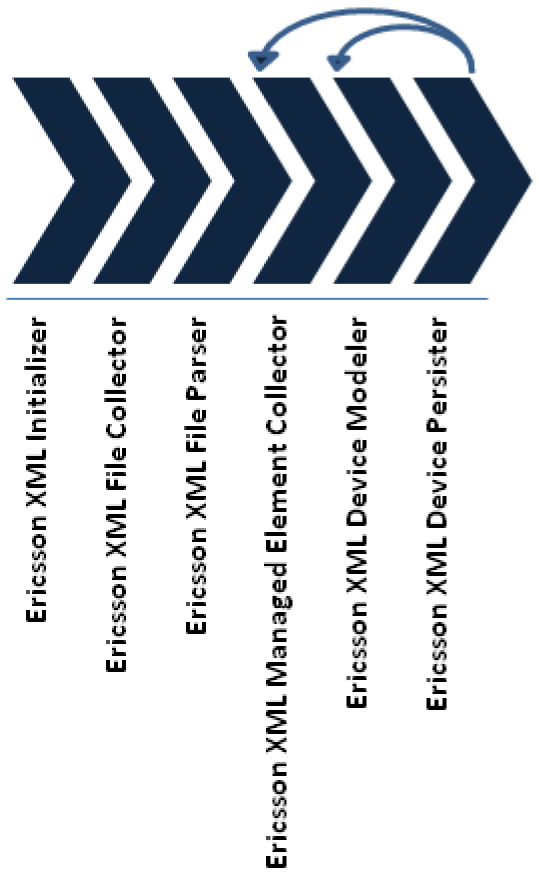
The Discover Ericsson Xml action reads one or more XML device file instances, and provides multiple hierarchical device model instances. (XML device file instances could contain multiple devices.)

The Discover Ericsson Xml action contains the following processors run in the following order:

1. Ericsson Xml Initializer
2. Ericsson Xml File Collector
3. Ericsson Xml File Parser
4. Ericsson Xml Managed Element Collector
5. Ericsson Xml Device Modeler
6. Ericsson Xml Device Persister

Figure 5–2 illustrates the processor workflow of the Discover Ericsson Xml action.

Figure 5–2 Discover Ericsson Xml Action Processor Workflow


Ericsson XML Initializer
Ericsson XML File Collector
Ericsson XML File Parser
Ericsson XML Managed Element Collector
Ericsson XML Device Modeler
Ericsson XML Device Persister

Ericsson Xml Initializer

The Ericsson Xml Initializer instantiates a helper class, and then makes it available to other processors in the chain.

Ericsson Xml File Collector

The Ericsson Xml File Collector processor is used to retrieve XML device files and make them available to the next processor in the chain.

Note: This processor is automatically generated from Design Studio input data.

Ericsson Xml File Parser

The Ericsson Xml file parser processor is used to read the XML device files, validate them against a loaded schema, and convert the XML device file to an XML bean, making it available for parsing to the next processor.

Note: This processor is automatically generated from Design Studio input data.

Ericsson Xml Managed Element Collector

The Ericsson Xml Managed Element Collector processor is used to process the XML device file and locate managed elements (MEs). These MEs are then inserted into a list for further processing.

Note: One or more managed elements are contained within nested subNetworks in the XML device file. This processor is capable of finding all managed elements. SubNetworks are not modeled.

Ericsson Xml Device Modeler

The Ericsson Xml Device Modeler processor is used to model the data collected from the Ericsson Xml Managed Element Collector. Modeling includes building the hierarchical relationship of physical device and children equipment and equipment holders from an individual ME.

Ericsson Xml Device Persister

The Ericsson Xml Device Persister is used to persist the physical device tree to the Network Integrity database.

About Collected Data

This section shows a sample XML device file that is provided to the processor.

```
<?xml version="1.0" encoding="UTF-8"?>
<bulkCmConfigDataFile xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.3gpp.org/ftp/specs/archive/32_
series/32.615#configData
../../../eclipseWorkSpace/XMLReferenceCartridge/schemas/configData.xsd"
   xmlns="http://www.3gpp.org/ftp/specs/archive/32_series/32.615#configData"
   xmlns:xn="http://www.3gpp.org/ftp/specs/archive/32_series/32.625#genericNrm"
   xmlns:in="http://www.3gpp.org/ftp/specs/archive/32_series/32.695#inventoryNrm">
   <fileHeader fileFormatVersion="32.615 V6.3" vendorName="Ericsson AB"/>
   <configData dnPrefix="DC=150.132.36.75,SubNetwork=NRO_
RootMo,ManagementNode=ONRM,IRPAgent=ONRM_IrpAgent">
      <xn:SubNetwork id="NRO_RootMo">
         <xn:SubNetwork id="RNC106">
            <xn:ManagedElement id="RNC106">
               <xn:attributes>
                  <xn:managedElementType>RNC</xn:managedElementType>
                  <xn:userLabel>RNC106</xn:userLabel>
                  <xn:vendorName>Ericsson AB</xn:vendorName>
               </xn:attributes>
               <in:InventoryUnit id="1B">
                  <in:attributes>
                     <in:inventoryUnitType>HW</in:inventoryUnitType>
                     <in:vendorUnitFamilyType>SUBRACK</in:vendorUnitFamilyType>
                     <in:vendorUnitTypeNumber>ROJ 605 107/3_
R1A</in:vendorUnitTypeNumber>
                     <in:vendorName>Ericsson AB</in:vendorName>
                     <in:serialNumber>X911033101</in:serialNumber>
                     <in:dateOfManufacture>2005-10-22</in:dateOfManufacture>
                     <in:unitPosition>1B</in:unitPosition>

<in:manufacturerData>ProductName=CBM,SlotCount=28</in:manufacturerData>
                  </in:attributes>
                  <in:InventoryUnit id="0">
```

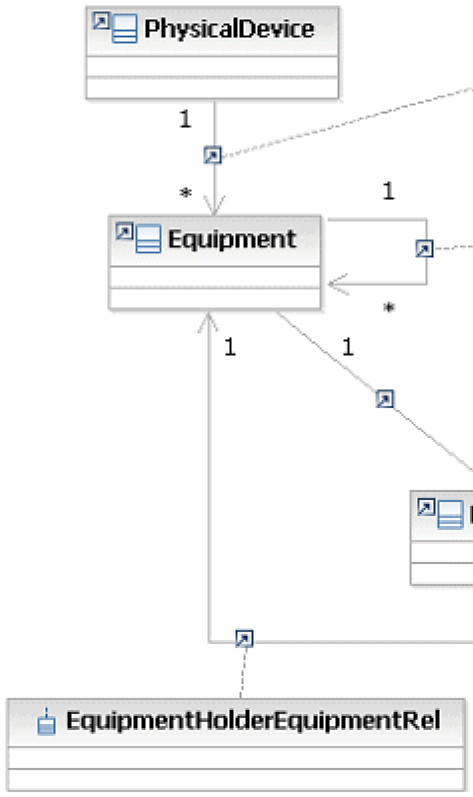
```
                     <in:attributes>
                        <in:inventoryUnitType>HW</in:inventoryUnitType>
                        <in:vendorUnitFamilyType>FAN</in:vendorUnitFamilyType>
                        <in:vendorUnitTypeNumber>BKV 301 487/1_
R3A</in:vendorUnitTypeNumber>
                        <in:vendorName>Ericsson AB</in:vendorName>
                        <in:serialNumber/>
                     </in:attributes>
                  </in:InventoryUnit>
                  <in:InventoryUnit id="1">
                     <in:attributes>
                        <in:inventoryUnitType>HW</in:inventoryUnitType>
                        <in:vendorUnitFamilyType>PIU</in:vendorUnitFamilyType>
                        <in:vendorUnitTypeNumber>ROJ1192108/4_
R2B</in:vendorUnitTypeNumber>
                        <in:vendorName>Ericsson AB</in:vendorName>
                        <in:serialNumber>TU87600308</in:serialNumber>
                        <in:dateOfManufacture>2005-12-01</in:dateOfManufacture>
                        <in:unitPosition>1</in:unitPosition>

<in:manufacturerData>ProductName=SCB3</in:manufacturerData>
                     </in:attributes>
                  </in:InventoryUnit>
                  <in:InventoryUnit id="2">
                     <in:attributes>
                        <in:inventoryUnitType>HW</in:inventoryUnitType>
                        <in:vendorUnitFamilyType>PIU</in:vendorUnitFamilyType>
                        <in:vendorUnitTypeNumber>ROJ1192109/3_
R1B</in:vendorUnitTypeNumber>
                        <in:vendorName>Ericsson AB</in:vendorName>
                        <in:serialNumber>TU87153523</in:serialNumber>
                        <in:dateOfManufacture>2005-10-07</in:dateOfManufacture>
                        <in:unitPosition>2</in:unitPosition>

<in:manufacturerData>ProductName=SXB3</in:manufacturerData>
                     </in:attributes>
                  </in:InventoryUnit>
                  <in:InventoryUnit id="3">
                     <in:attributes>
                        <in:inventoryUnitType>HW</in:inventoryUnitType>
                        <in:vendorUnitFamilyType>PIU</in:vendorUnitFamilyType>
                        <in:vendorUnitTypeNumber>ROJ1192109/3_
R1B</in:vendorUnitTypeNumber>
                        <in:vendorName>Ericsson AB</in:vendorName>
                        <in:serialNumber>TU87153427</in:serialNumber>
                        <in:dateOfManufacture>2005-10-07</in:dateOfManufacture>
                        <in:unitPosition>3</in:unitPosition>

<in:manufacturerData>ProductName=SXB3</in:manufacturerData>
                     </in:attributes>
                  </in:InventoryUnit>
               </in:InventoryUnit>
            </xn:ManagedElement>
         </xn:SubNetwork>
      </xn:SubNetwork>
   </configData>
   <fileFooter dateTime="2006-09-06T08:03:04+02:00"/>
</bulkCmConfigDataFile>
```

About Cartridge Modeling

This section provides information about modeling the XML Reference cartridge.

Figure 5–3 shows a Unified Modeling Language (UML) diagram depicting the object relationship.

Figure 5–3 XML UML Diagram



Hierarchy Mapping

The physical device object is established and seeded with data sourced by ManagedElement.

The Equipment object is established and seeded from InventoryUnit, where the vendorUnitFamilyType = PIU (Plug In Unit). All other InventoryUnits are discarded, that is (vendorUnitFamilyType = OTHER, FAN).

The EquipmentHolder object is established and seeded from InventoryUnit, where the vendorUnitFamilyType =SUBRACK.

Note: Modeling limits InventoryUnit to two (2) levels deep. If an XML file instance has more than two levels of InventoryUnit below ManagedElement, the third to nth levels are ignored, and customization is required to model those levels.

Oracle Communications Information Model

All entities shown in Figure 5–3 (for example, physical device and equipment) are Information Model 1.0-compliant for static fields. The dynamic fields (sometimes referred to as characteristics) are application-specific.

Field Mapping

This section provides information about field mappings used in the cartridge.

- Text: Implies Text [255].
- static: The Information Model 1.0 defines this field to be static on the entity specification. The specification provides getters/setters for this field.
- dynamic: This is a dynamic field where the entity specification treats the field as a name/value pair. The specification does not provide getter/setters but generically has a get/setCharacteristics method holding a HashSet of entries.

Table 5–1 Physical Device Mappings

Physical Device	Information Model Support	Xml Object	Field Type
Id	static	N/A	Text
Name	static	id	Text
Description	static	xmlFile Source	Text
Specification	static	N/A	Programmatically set to EricssonCPPPhysicalDevice
discoveredVendorName	dynamic	vendorName	Text
modelName	dynamic	managedElementType	Text
nativeEmsName	static	userLabel	Text
Serial Number	yes	N/A	Text
Physical Location	yes	N/A	Text

Table 5–2 Equipment Mappings

Equipment	Information Model Support	XML Object	Field Type
Id	static	N/A	Text
Name	static	manufacturerData	Text, Extract ProductName
Description	static	manufacturerData	Text
Specification	static	N/A	Programmatically set to EricssonCPPEquipment
discoveredModelNumber	dynamic	vendorUnityType Number	Text
discoveredVendorName	dynamic	vendorName	Text
serialNumber	static	serialNumber	Text
nativeEmsName	static	manufacturerData + unitPosition	Text, Extract ProductName, append unitPosition which is occupied slot number

Table 5–2 (Cont.) Equipment Mappings

Equipment	Information Model Support	XML Object	Field Type
Physical Location	yes	N/A	Text

Table 5–3 EquipmentHolder Mappings

EquipmentHolder	Information Model Support	XML Object	Field Type
Id	static	N/A	Text
Name	static	manufacturerData + slotNumber	Text
Specification	static	N/A	Programmatically set to EricssonCPPEquipmentHolder
nativeEmsName	static	slotNumber	Text manufacturerData contains SlotCount. SlotCount is used to generate the required number of slots and slotNumber is a slot instance.
Description	yes	N/A	Text
Serial Number	yes	N/A	Text
Physical Location	yes	N/A	Text

Model Correction

This section provides 3GPP to Information Model correction information.

About Model Correction Code

Model correction occurs when the 3GPP information received through discovery does not conform to the Information Model and therefore cannot be persisted, as it is within Network Integrity. For information about supported hierarchy, see "About Cartridge Modeling".

The XML Reference Cartridge applies the model corrections as outlined below.

EquipmentHolder under physical device:

```
PhysicalDevice
   EquipmentHolder
```

The XML Reference Cartridge adds an equipment entity as follows:

```
PhysicalDevice
  Equipment-named Artificial Equipment
   EquipmentHolder
```

Design Studio Construction

This section provides information about using Design Studio to construct the XML Reference cartridge.

The XML Reference cartridge contains the following specifications:

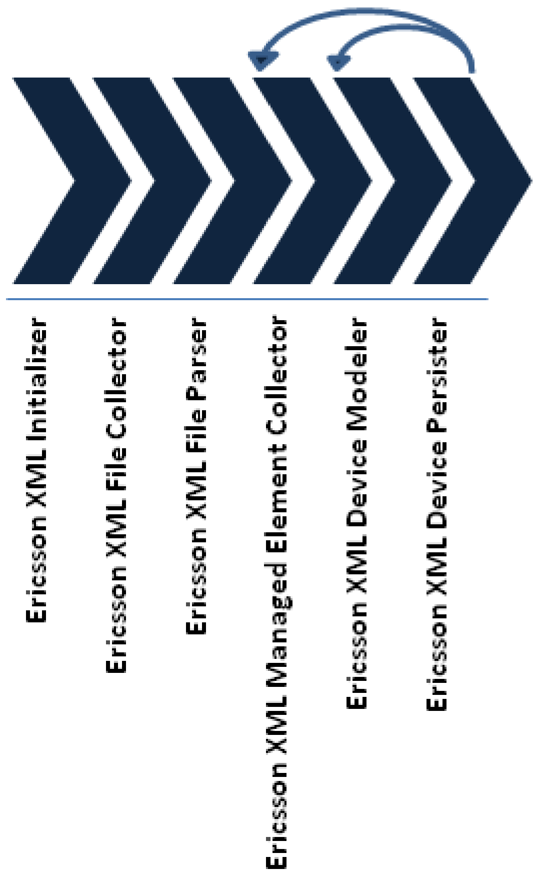
- EricssonCPPPhysicalDevice
- EricssonCPPEquipment
- EricssonCPPEquipmentHolder
- EricssonXmlFileCollectorProperties

Table 5–4 Discover Ericsson Xml Action Construction

Result Category	Address Handler	Scan Parameter Groups	Model	Processors
Device	FileTransfer AddressHandler	ftaFileTransferType ftaFilePattern ftaPort ftaUser ftaPassword ftaSessionTimeOut ftaSourceFileManagement ftaRenameSuffix	Ericsson CPP Model	Ericsson Xml Initializer Ericsson Xml File collector Ericsson Xml File Parser Ericsson Xml Managed Element Collector Ericsson Xml Device Modeler Ericsson Xml Device Persister

Figure 5–4 shows the discover DiscoverEricssonXml action chain.

Figure 5–4 Discovery XML Action Chain


Ericsson XML Initializer
Ericsson XML File Collector
Ericsson XML File Parser
Ericsson XML Managed Element Collector
Ericsson XML Device Modeler
Ericsson XML Device Persister

In Figure 5–4, the chevrons that correspond to EricssonXmlFileCollector and EricssonXmlFileParser indicate the code-generated processors from Design Studio user input.

- EricssonXmlFileCollector is an instance of the file transfer processor
- EricssonXmlFileParser is an instance of the file parser processor

Table 5–5 Discover Ericsson Xml Action Processors

Processor Name	**Variable**
Ericsson Xml Initializer	Input: N/A Output: ■ physicalDeviceHelper The output is a helper class used in proceeding chain.
Ericsson Xml File Collector	Input: N/A Output: ■ ericssonXmlFileCollectorFileCollection <java.util.Collecton> The output is a collection of files found in the path specified in the scope field.
Ericsson Xml File Parser	Input: ■ ericssonXmlFileCollectorFileCollection The input is a collection of files. ■ XmlSchema: schemas/configData.xsd The schema is used to validate XML file instances. Output: N/A
Ericsson Xml Managed Element Collector	Input: XmlFile Output: ■ managedElements <java.util.ArrayList> The output is a list containing the managedElements. The processor is wrapped in a For each loop to execute this processor for each XML file.
Ericsson Xml Device Modeler	Input: ManagedElement, physicalDeviceHelper Output: physicalDevice The processor is wrapped in a For each loop to execute this processor for each ME. This processor can be extended to enhance an individual physical device tree.
Ericsson Xml Device Persister	Input: N/A Output: N/A Context is persisted for performance.

Note: configData.xsd is the root document of the XSD formed from documents configData.xsd, genericNrm.xsd, inventoryNrm.xsd, and sessionLog.xsd. These documents had to first be stitched and validated after retrieval from:

`http://www.3gpp.org`

See the "The ASCII Reference Cartridge" for 3GPP URLs. Stitching implies that schemaLocation had to be input into the configData.xsd and inventoryNrm.xsd. In addition, all unused namespaces are removed.

Design Studio Extension

This section provides information about Design Studio extensions to the XML Reference cartridge.

The source code to this cartridge is provided. You can change any part to customize this cartridge to fit your environment.

For more information on extensibility, see *Network Integrity Developer's Guide*.