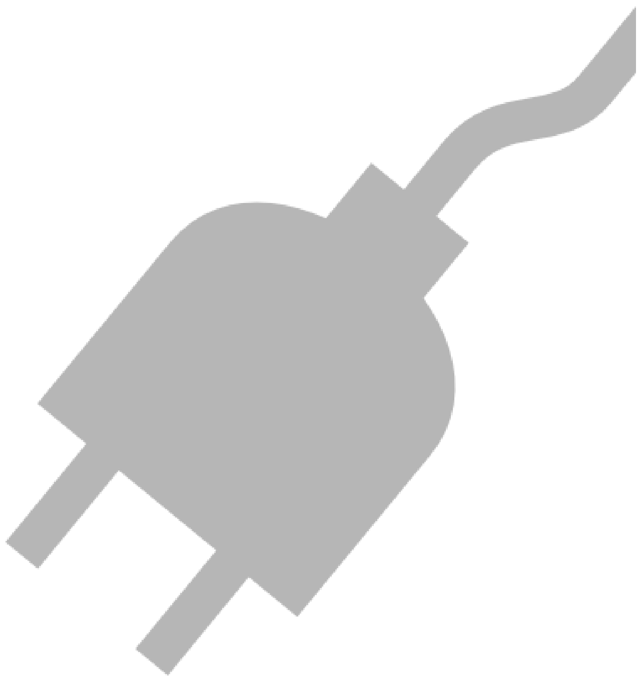
EXTDATA Plug-in User's Manual

Symbolic Nuclear Analysis Package (SNAP)



Version 2.2.0 - August 02, 2021

Applied Programming Technology, Inc.

240 Market St., Suite 301
Bloomsburg PA   17815-1951

EXTDATA Plug-in Users Manual

Applied Programming Technology, Inc.
by Ken Jones and Dustin Vogt

Table of Contents

Chapter 1. Introduction

The EXTDATA Plug-in provides the ability to import external data sources into SNAP for animation. External data could be obtained from data aquisition systems used to record experimental data or any type of calculated data. The data must first be converted into a Platform Independent Binary (PIB) format. The file contains a file header section, followed by an optional section containing engineering units conversions, and then the experimental data listed by time slice. Engineering units are defined for each data channel using either an NRC Databank engineering units code or a reference to an embeded engineering units conversion.

The EXTDATA plug-in is also a library that can be used to write EXTDATA files. The class ExtDataWriter allows a simple means of writing experimental data in the proper format.

The NRC DB Data Extractor tool is bundled with the plug-in. This application allows creating EXTDATA ready data files from NRC Databank platform independent binary files.

Chapter 2. Using EXTDATA Files

To utilize EXTDATA files, you will need to perform the following steps in JobStatus (or any SNAP window with similar functionality).

1. Navigate to a mounted location on a running server containing an EXTDATA file (such as extdata.pib, exported by the Data Extraction tool).

2. Right-click on the folder containing the file and select **Import Local File** from the pop-up menu.

3. In the ensuing dialog, set *PLUG-IN* to **EXTDATA**, *Job Name* to an appropriate name for the imported file, and *Local File* to the EXTDATA file itself.

The file is now listed in the folder as a completed job, suitable for use in post-processing (such as animation).

Note Unlike early versions of the plug-in, the EXTDATA file need no longer be identified only as `extdata.pib`, and any number of EXTDATA files may reside alongside each other in a mounted folder.

Note The plug-in is distributed with two sample files that can be used to demonstrate EXTDATA functionality: `extdata.pib` and `sample.med`. Once `extdata.pib` is imported as a local file, it can be used as a Data Source in `sample.med` and animated.

Chapter 3. Running the Data Extractor Tool

The Data Extractor utility can be used to convert several forms of data files directly to EXTDATA files. The method of launching the application varies slightly depending on the user's platform. The following are all valid:

- On Windows, a Data Extractor item is added to the Start Menu in the SNAP folder.
- For *NIX systems with GNOME or KDE, the installation creates similar menu items.
- All other users can launch the application directly through the "dataextractor" launchers found in the SNAP installation's `bin` folder.

Data Extractor usage and features are detailed more thoroughly in the application's own documentation.

Chapter 4. EXTDATA File Format

The EXTDATA_PLUGIN expects the data file to be an XDR binary file conforming to the following format. Various housekeeping information is written to the file, but it has the following basic structure: a file header, an optional code listing, and a block of data for each slice of time in the experiment.

Note Unlike early versions of the plug-in, the EXTDATA file need no longer be identified only as `extdata.pib`, and any number of EXTDATA files may reside alongside each other in a mounted folder.

The file header is always expected to be the first block of information in the file (to that end, an ExtDataWriter throws an exception if a program attempts to write a block of data before writing the file header). This header tracks a number of values tied directly to the plug-in's read methods. It is generally absolutely necessary for the file header to contain accurate information. It stores the following values:

- *`Facility`*. Where the experimental data was recorded.
- *`Experiment`*. The experiment to which this data corresponds.
- *`Description`*. A description of the experiment.
- *`Unit Source`*. The source of this data's engineering unit codes.

  Note Unit Source has replaced the NRC DB EU Codes flag of the previous version of the plug-in.
- *`Experiment start time`*. Start time for the experiment.
- *`Experiment end time`*. End time for the experiment.
- *`Number of time slices`*. The number of time slices between the start time and the end time, inclusive.
- *`Channel headers`*. Explained below.

A channel header describes a data channel. The file header stores a sequential listing of all channel headers in the experiment. The first channel must be time. Channel headers store the following information:

- *`Name`*. The name of the data channel.
- *`EU Code`*. The engineering unit code. This code directly corresponds to a conversion set specified by the Unit Source. Normally, this refers to NRC Databank EU codes, but may also directly reference custom units embedded directly in the file.

A file is expected to have an engineering unit code listing if the file header indicates that the data is to utilize an embedded code listing. Each unit code has the following information:

- *`Engineering Unit Code`*. The index of this engineering unit code. This value must be non-negative and unique among EU codes in the file.
- *`Unit Description`*. A short description of the unit code, usually the name.
- *`SI Label`*. Label for this unit in the SI system of measure (ex: m for meters, N for newtons, etc.).
- *`British Label`*. Label for this unit in the British system of measure (ex: ft for feet, lb for pounds, etc.).
- *`Storage Type`*. Integer tracking whether the value written to the file is stored in SI or British units. For SI values, this must be 0; for British values, this must be 1.
- *`Conversion Factor`*. The conversion factor, explained more below.
- *`Conversion Offset`*. The conversion offset, explained more below.

The conversion factor and offset merit some explanation, as these have changed in semantics from legacy versions of the plug-in where custom unit codes could be written, but not utilized directly in SNAP. The conversion factor and offset are used such that values can be converted from practically any form of measurement, including scales with arbitrary offsets, such as temperature. The factor and offset are always interpreted in very specific means and order: the conversion values must directly convert SI values to British units. The equations used to perform the conversions are as follows:

$$\mathrm{Value}_{\mathrm{British}} = (\mathrm{Value}_{\mathrm{SI}} * \mathrm{Factor}) + \mathrm{Offset}$$

$$\mathrm{Value}_{\mathrm{SI}} = (\mathrm{Value}_{\mathrm{British}} - \mathrm{Offset}) / \mathrm{Factor}$$

These are consistent with the above requirement that the factor and offset directly transform SI values to British units.

The final segment of the file will usually make up the bulk of it: data for time slices. Each data block is an array of double-precision floating point numbers. The array length must exactly match the number of data channels listed in the file header. The first element of the data block is always expected to be the time for that slice of data. The elements of the data block must be in order corresponding to the data channels listed in the header. These data slices must be written in sequential order according to time.

Chapter 5. ExtDataWriter

The ExtDataWriter class has been written to streamline the process of writing data to a PIB file. Instances will throw ExtDataWriterException (a checked exception) for a variety of failures. Its public methods are as follows:

- `ExtDataWriter()`. Create an intance of ExtDataWriter.
- `ExtDataWriter(String filename)`. Create an instance of ExtDataWriter with the target file specified by the parameter open for writing.
- `boolean isOpen()`. Returns true if a target file is open. Returns false otherwise.
- `boolean isHeaderSet()`. Returns true if a file has been opened and its file header has been written. Returns false otherwise.
- `boolean isUsingNrcCodes()`. Returns true if a file has been opened, its file header has been written, and the file is expected to use the NRC Databank engineering unit codes. Returns false otherwise.
- `boolean isOtherCodeSet()`. Returns true if a file has been opened, its file header has been written, the file is not expected to use the NRC Databank engineering unit codes, and the alternative code list has been written. Returns false otherwise.
- `String getUnitSrc()`. Returns the unit source of the data file if the file header has been written. Current values include: *`ExtDataWriter.NRCDB`*, `ExtDataWriter.EMBEDDED`, and *`ExtDataWriter.RELAP5`*. Returns null otherwise.
- `void openFile(String fileName)`. Opens a new file for writing. Closes a preexisting open file.
- `void writeFileHeader(String facility, String experiment, String description, String unitSrc, double startTime, double endTime, int numTimeSlices, String[] channelNames, int[] euCodes)`. Writes a file header with the given information. The array arguments must be parallel and equal length; the first element is expected to be time. The other parameters correspond to the above mentioned values. Various constants have been provided for unitSrc in the ExtDataWriter class: `NRCDB`, `EMBEDDED`, and `RELAP5`. Other constants may be added in a future release.

  Throws `ExtDataWriterException` if no file is open, if the file header has already been written, if channelNames and euCodes do not have matching length, or if the write fails.

  Note *`String unitSrc`* has replaced *`boolean nrcCodes`* of legacy versions.

- `void writeDataBlock(double data[])`. Writes a block of data pertaining to one time slice to the target file. The argument data must be at least as large as the expected number of data channels, preferably exactly equal and corresponding in order to the data channels specified in the file header; excess elements are discarded.

  Throws `ExtDataWriterException` if no file is open, the header has not been written, the file does not intend to use the NRC Databank Engineering Unit Codes and no code list has been written, data has less values than expected, or if the write fails.

- `void writeData(double data[][])`. Writes several blocks of data to the target file. The two dimensional array passed as the argument should have the following form: the first dimension corresponds to the time slices, the second dimension corresponds to the values for that time slice, where the first element should be time. data must be at least as wide as the expected number of channels, preferably exactly equal and corresponding in order to the data channels specificed in the file header; excess elements are discarded.

  Throws `ExtDataWriterException` if no file is open, the header has not been written, the file does not intend to use the NRC Databank Engineering Unit Codes and no code list has been written, data has less values in its second dimension than expected, or if the write fails.

- `void writeCodeList(UnitCode[] codes)`. Writes a list of engineering unit codes to the target file if the data is not intended to utilize the NRC Databank Engineering Unit Codes. This method should not be called until a file header is written with its unitSrc set to ExtDataWriter.EMBEDDED, and must be called in that event before any data is written.

  Throws `ExtDataWriterException` if the file is not open, the header has not been written, the file is not marked as using an embedded code listing, the code list has already been written, or the write fails.

- `void close()`. Closes the file. Should be called explicitly when the ExtDataWriter is no longer of use.

5.1. Code Sample for ExtDataWriter

The following is a code sample detailing the use of the ExtDataWriter class. This file is also included with the plug-in installation in the `Samples` folder.

```
// Copyright (c) 2002-2005 Applied Programming Technology, Inc.
// All rights reserved.

import com.cafean.extdata.io.ExtDataWriter;
import com.cafean.extdata.io.ExtDataWriterException;

/**
 * A sample program of how to use the ExtDataWriter class for writing data to a
 * file.
 *
 * @author  APT, Inc.
 */
public class ExpFileSample
{
    public static void main(String[] args)
    {
        double currentTime = 0.0;
        double timeIncrement = 1.0;

        String facility = "APT, Inc.";
        String experiment = "Example Experiment";
        String description = "Sample data for ExtDataWriter example.";
        // Current options are ExtDataWriter.NRCDB, ExtDataWriter.EMBEDDED, and ExtDataWriter.RELAP5
        String unitSrc = ExtDataWriter.NRCDB;
        double startTime = 0.0;
        double endTime = 9999.0;
        int numTimeSlices = 10000;
        // Note that the end time might be expected to be an exact value of
        // 10.0. In this example the actual end time generated by incrementing
        // 0.0 by 0.1 a thousand times is actually closer to 9.98 due to round
        // off error. It is up to the user to insure that the set end time and
        // the actual end time agree.

        // Channel headers to be stored in the file header.
        String[] cHeaderNames = new String[5];
        int[] cHeaderCodes = new int[5];
        cHeaderNames[0] = "Time";
        cHeaderCodes[0] = 36; // Corresponds to time in the NRC Databank EU Codes.
        cHeaderNames[1] = "Frequency";
        cHeaderCodes[1] = 56;
        cHeaderNames[2] = "Current";
        cHeaderCodes[2] = 11;
        cHeaderNames[3] = "Pressure";
        cHeaderCodes[3] = 3;
        cHeaderNames[4] = "Mass";
        cHeaderCodes[4] = 46;

        // Open the file.
        ExtDataWriter dataWriter = new ExtDataWriter("extdata.pib");

        // Write the file header.
        try {
            dataWriter.writeFileHeader( facility, experiment, description,
                                        unitSrc, startTime, endTime,
                                        numTimeSlices, cHeaderNames,
                                        cHeaderCodes );
        } catch(ExtDataWriterException e) {
            e.printStackTrace();
            return;
        }

        // If the data is not to be tied to the NRC databank engineering unit
        // codes, a CodeList must now be written to the file.
        // Note: a more practical use would have to include a definition for Time.
        //       This sample is more interested in demonstrating units with conversions.
//        UnitCode[] uCodes = new UnitCode[2];
//        uCodes[0] = new UnitCode();
//        uCodes[0].setEuCode(1); // All EU code values must be both unique and non-negative.
//        uCodes[0].setEuType(0); // 0 for values written to the file in SI, 1 for British
//        uCodes[0].setUnitDescription("Length");
//        uCodes[0].setSiLabel("cm");
//        uCodes[0].setBrLabel("in");
//        uCodes[0].setConversionFactor(0.393700787);
//        uCodes[1] = new UnitCode();
//        uCodes[1].setEuCode(2);
//        uCodes[1].setEuType(0);
//        uCodes[1].setUnitDescription("Temperature");
//        uCodes[1].setSiLabel("C");
//        uCodes[1].setBrLabel("F");
//        uCodes[1].setConversionFactor(1.8);
//        uCodes[1].setConversionOffset(32);

        // A note about conversions. Units can be entered to convert between
        // practically any units of measurement, including scales with offsets,
        // such as temperature. However, for the sake of clarity, all
        // conversions must be entered so that they convert from SI units to
        // British units, regardless of which type of unit is used to store
        // the value in the file.
        //
        // The conversion formulae used are:
        //    SI to British: (value * factor) + offset
        //    British to SI: (value - offset) * factor
        // These expressions are consistent with the requirement that all
        // conversion values translate from SI to British.

        // The following writes the units list.
//        try {
//            dataWriter.writeCodeList(uCodes);
//        } catch(ExtDataWriterException e) {
//            e.printStackTrace();
//            return;
//        }

        // Generate data and write data.
        for(int i = 0; i < numTimeSlices; i++) {
            // Can only have as many elements as data channels.
            double exampleDataSlice[] = new double[5];
            // Generate dummy data.
            exampleDataSlice[0] = currentTime;
            exampleDataSlice[1] = Math.sin((double)i/10.0);
            exampleDataSlice[2] = Math.cos((double)i/10.0);
            exampleDataSlice[3] = Math.sin((double)i/5.0);
            exampleDataSlice[4] = Math.cos((double)i/5.0);
            currentTime += timeIncrement;

            // Write data.
            try {
                dataWriter.writeDataBlock(exampleDataSlice);
            } catch(ExtDataWriterException e) {
                e.printStackTrace();
                return;
            }
        }

        /* Alternatively, the following could have been used:
         *
         * double exampleData[][] = new double[1000][4];
         * for(int i = 0; i < exampleData.length; i++) {
         *     // Generate data.
         *     exampleData[i][0] = currentTime;
         *     exampleData[i][1] = Math.sin((double)i/10.0);
         *     exampleData[i][2] = Math.cos((double)i/10.0);
         *     exampleData[i][3] = 2*Math.sin((double)i/5.0);
         *     currentTime += timeIncrement;
         * }
         *
         * // Write data.
         * try {
         *     dataWriter.writeData(exampleData);
         * } catch(ExtDataWriterException e) {
         *     e.printStackTrace();
         *     return;
         * }
         */

        // Housekeeping.
        dataWriter.close();
    }
```

```
        }
```

Index