Getting Started with SDDS

Version 0.1

Michael Borland
Advanced Photon Source
Argonne National Laboratory
borland@aps.anl.gov

Introduction

SDDS, or Self Describing Data Sets, is a way of storing and working with data that was developed at the Advanced Photon Source (APS) for use in the simulation and operation of accelerators. Because SDDS is very generic in nature, it can be used for processing and displaying data from essentially any source. This document describes the concept behind SDDS, the implementation of that concept, the capabilities of implementation, as well as problems and limitations. Numerous examples are given to guide the reader in using SDDS and developing applications based on SDDS.

Parts of SDDS are linked to the Experimental Physics and Industrial Control System (EPICS), which is used worldwide to control particle accelerators, telescopes, and other scientific equipment. At APS, we use SDDS and the Tcl/Tk scripting language to develop graphical user interfaces (GUIs) for controlling our accelerators. This includes configuration of the accelerators; data collection, analysis, and display; experiment execution; and feedback processes, among others.

In addition to being used for control system applications, SDDS is used for pre− and post−processing of data for accelerator simulations. Again, since SDDS is generic in nature, it can be used with other types of computer simulations and data sources. Using SDDS for computer simulations has several advantages. Principle among these is that one's simulation programs can be simplified without loss of overall capability. SDDS provides tools to manipulate the data that serves as input to such programs, so that the programs themselves do not need to support such operations. SDDS also provides a highly capable system for analysis and display of simulation output. Hence, the simulation code doesn't need to perform graphics or provide a general−purpose data analysis facility. SDDS also makes input and output programming less problematical by providing a set of programming tools for verifying and reading input data and preparing and writing output data.

SDDS is based on two concepts: the use of "self−describing data" and the use of a "toolkit" of programs that operate on self−describing data. Self−describing data is simply data that is identified and accessed by name only. An example of data that*isn't* self−describing is data in a typical spreadsheet. Very commonly, columns of data in a spreadsheet are unlabeled and have meaning only because the user remembers what columns A, B, and so on contain. If the user adds a row of headings to the spreadsheet, then the data has acquired a basic self−describing character. In principle, any program could read the data file and determine what data it contains.

Using a spreadsheet file as a general self−describing file has several problems, however. Spreadsheet files are amorphous in character. The user is not constrained in how the data is laid out, nor is he required to provide labels for each data element. Hence, it is possible for the user to accidentally or deliberately create a non−self−describing file or an oddly formatted file. While this is not necessarily a problem in the context of using a spreadsheet, it creates difficulties if one wants to use a set of generic tools for manipulation of this file and other similar files. General spreadsheet data is only guaranteed to be readable by the spreadsheet program itself.

In contrast, true self−describing data can in principle be read equally by any number of programs, including programs written by the user. No particular program "owns" the self−describing file in the way that a spreadsheet program owns a spreadsheet file or a wordprocessing program owns a document. The spreadsheet program is only one example of a program that has a specific, essentially private format for its files. It is very common for custom−designed software in the scientific community to make use of application−specific file formats that are typically readable only by specially−written programs that read only the files from that program. This includes not only programs that operate in a control system environment, but also simulation programs.

With self−describing data, the approach is fundamentally different: Instead of defining our file format with respect to the programs that create or use them, we define a standard file protocol and design all programs to work with such files.

Let’s return to the discussion of self−describing data. When accessing data from a self−describing file, a program asks for the data using a procedure library that allows it to specify the name of the data and its class. For example, a program might request the item in the "array" class called "Temperatures" or the item in the "parameter" class called "Date". The program will be able to determine whether the desired data actually exists, what its data type is (e.g., floating point or character string), as well as optional information such as units, description, and dimensions.

Contrast this again with non−self−describing data, which provides none of these features. With such data, one cannot *determine* whether the desired data exists in the file, but must *assume* it. Similarly, one cannot *determine* the data type, class, or units. This inability to verify what the data is makes the program more likely to fail, crash, or do something inappropriate. So we see that there are some clear advantages to using self−describing data from the standpoint of program reliability.

Many programmers naively assume that this isn’t a problem, believing they know what their data will look like and that they can build that knowledge into their applications. The difficulty with this approach is that software is always changing. It is very common for an application or program to evolve steadily over the period of its usefulness. In fact, it is likely that when it stops changing, it is because it it is no longer in demand and is going to be supplanted by something more capable. Hence, it is important to realize that software will be required to evolve and to use file protocols that make this easy.

In the process of evolving one’s software, one should strive to maintain backward compatibility for old data files and interoperability with other applications. This is possible using self−describing data. For example, an upgraded application can detect the absence of new data elements (signifying an old data file) and supply a default behavior. At the same time, when the new data elements are present, the upgraded application can

behave appropriately. Other applications that make use of the same data files will simply ignore the additional data elements until or unless they are also upgraded. Hence, one need not upgrade all applications at the same time. This stands in stark contrast to the common situation in physics, where modification of a simulation code's input or output data requires simultaneous modification of several other codes, and makes old input and output files unusable.

This discussion is relevant here because SDDS files are self−describing and hence applications that use SDDS files enjoy all of the advantages just discussed. However, there is more to SDDS than this. SDDS is not only a self−describing file protocol, it is also a set of tools for manipulating such files. These tools are *programs*, as opposed to *applications*, by which I mean that they perform a single function that is not necessarily useful by itself. The SDDS Toolkit programs are used to create or support applications that enjoy the advantages of using self−describing files.

The word "toolkit" is used pretty frequently these days, and often inapprorpiately to refer to a grab−bag of unrelated programs. We all know that physical tools are all related in the sense that they can be used sequentially to perform a job that is impossible with any single tool. Hence, one can build a house with a hammer and a saw together, but not with a hammer only. The programs that make up the SDDS Toolkit are like real tools in that they can be used sequentially on the same object, in this case an SDDS data file. Like real tools, the usefulness of each SDDS tool is amplified tremendously by this interoperability. Indeed, some tools that would be completely useless by themselves are in fact tremendously useful because of the existence of a cooperative set of programs. This provides an amplified return on one's programming effort.

To understand how the SDDS Tools are used, it helps to use an analogy from mathematics. One can write a general data processing algorithm as an equation in the form $R=O_n...O_2O_1I$. In this operator equation, O_n through O_1 represent mathematical operators that are applied successively to the input object I , in order to produce result R . With SDDS, we can do the same thing using programs. The "input" object is just an SDDS file, as is the result of the application of each "operator", or program. The SDDS file is a container object. The programs modify the contents of this object but not the type of object. This permits the programs to be combined in an essentially arbitrary fashion. Just as in mathematics, one can compose very general data processing algorithms using a sequence of SDDS programs. In addition, just as in mathematics, each program (operator) can be used in multiple applications (algorithms). This is another factor that increases the payoff from the creation of each program.

Some examples may serve to make all this a little clearer. I'll draw these examples from the accelerator field, since that's what I'm most familiar with. In an accelerator like the APS ring, we commonly store a beam of electrons for hours at a time. The beam intensity slowly decays, following an exponential decay law, at least for short periods of time. A common requirement at such facilities is to determine the "beam lifetime," which is just the exponential decay time. The algorithm for making this determination can be written as a series of operations: **acquireData | takeLog | fitLine | display.** In this "pseudo−command," I'm representing operators by words describing the operations. Each operator delivers data to the next operator, via a "pipe," represented with the vertical bar. In words, we first acquire some data (beam current vs time), then compute

the log of the current. Following that, we fit a line, then display the results.

This simple algorithm will work quite nicely when the data is of good quality. Suppose, however, that we try this and find that the fit is not always good due to noisy data or occassional bad data points. In that case, we could modify the algorithm slightly: **acquireData | takeLog | fitLine | removeOutliers | fitLine | display.** I've added two operations to the algorithm. The first new operation removes "outlier" data points, i.e., those that are not well characterized by the linear fit. Having removed the outlier data points, we then repeat the fit using only the good data points. The result is more reliable and less subject to noise and instrument errors.

This example illustrates two important advantages of the operator−based approach to developing algorithms. First, we can quickly test an idea for an algorithm and see how it performs. We are *using* programs rather than *writing* programs, with all the time savings that implies. Second, having tested the basic idea, we can insert or append new operators to improve and extend the results. Using this approach allows us to easily see the results of each change and judge whether it meets our needs or not.

Very frequently, those who work with SDDS Tools use temporary data files when developing an algorithm. The first step is usually to collect some data:**acquireData > tmpFile1.** The data is then plotted to get a feeling for what kind and quality of signals are present. Following this a series of operations are performed on the data and the results displayed after each operation. Once a satisfactory sequence has been worked out, it is then collected into a single command with operations connected by pipes.

This approach has a number of advantages over the more common approach of writing data processing algorithms in a programming or scripting language. First, when using SDDS one need not define and keep track of variables, arrays, looping indices, and so on. Second, one doesn't have to endure the edit, compile, debug, test cycle. Third, one can easily examine the results of each processing step, without the extra work of adding statements to one's code to provide intermediate output. Fourth, since most SDDS Toolkit programs process data based on wildcards, one can often specify large numbers of operations with very brief commands. The net result of these advantages is that once one has gained familiarity with SDDS, one can usually process and display one's data in a fraction of the time that would normally be required. This has repeatedly been our experience at APS.

The SDDS File Protocol

Up until now, I've discussed self−describing data files in a general way. In this section, I'd like to give more specifics about the SDDS file protocol. It isn't necessary that the user understand the internal details of SDDS files. However, it is important to understand the SDDS "data model" in order to make effective use of the SDDS Toolkit. The "data model" is simply the general form of the data that can be stored in a particular file protocol. While it is possible to have several data models for a given file protocol, SDDS has only one.

The SDDS data model was developed with the operator concept (discussed in the last section) in mind. This was one motivation for having a single data model and making it relatively simple. If the data model had been too general, it would have been too complicated to create and use the operator approach. Another motiviation was that a

simple data model will in fact accommodate most types of data naturally and easily, so that the simple data model makes the user's task easier in most cases. Those data sets that are not easily accommodated by the SDDS data model can be broken into several data sets in separate files. This is a small price to pay given that it greatly reduces the complexity of developing and using the SDDS Toolkit in the majority of cases.

The design of SDDS data model started with the realization that a great deal of scientific data can be naturally organized into a table with named columns, together with a set of named scalar values ("parameters") that pertain to the table. For example, weather data for a series of cities could be organized as shown in the following diagram:

Average Temperature (deg C)	***Average Humidity (%)***	***Warmest City***
35	68	Los Angeles

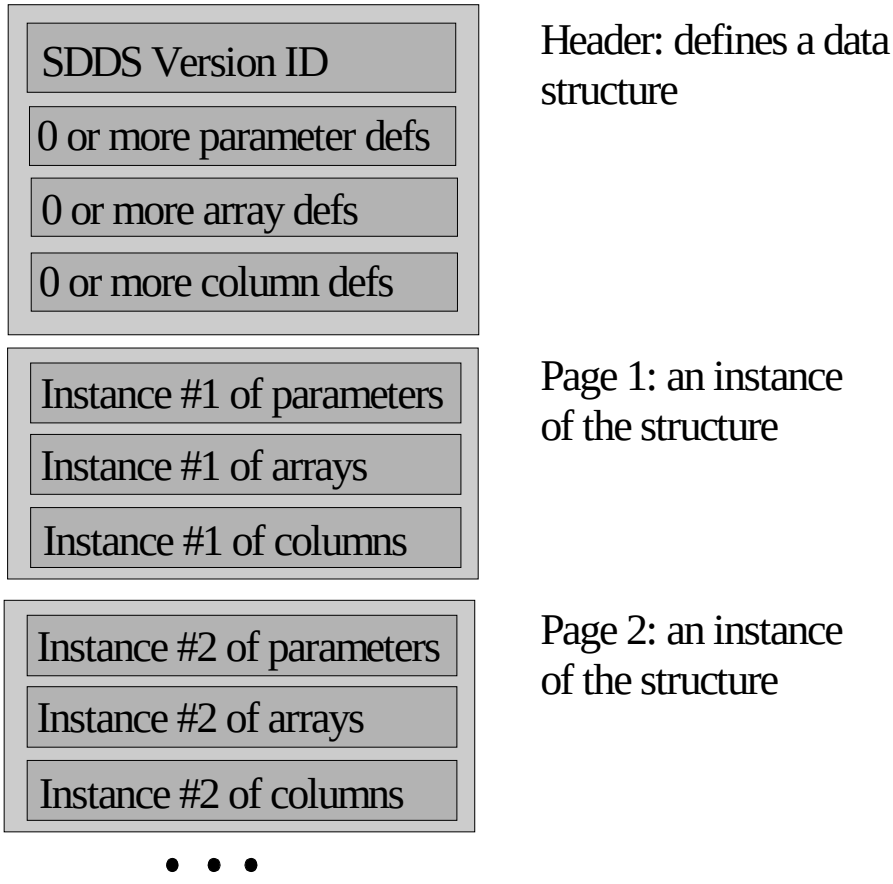
City	***Temperature (deg C)***	***Humidity (%)***
Chicago	38	95
New York	32	70
San Francisco	30	42
Los Angeles	40	65

In the language of the SDDS data model, "Average Temperature", "Average Humidity", and "Warmest City" are "parameters," while "City", "Temperature", and "Humidity" are "columns." The set of data for the columns is referred to as the "tabular data." The parameters in this case are a summary or abstract of the tabular data. This is typical of how parameters and columns are used together, and a relationship that is supported by the SDDS Toolkit. For example, if the tabular data above were in an SDDS file, the parameter values could be easily added using the program**sddsprocess.**

In addition to parameters and columns, the SDDS data model supports another type of element, namely, the array element. This allows placing any number of N−dimensional arrays in the file, where N is potentially different for each array. Clearly, any data that can be stored in parameters or columns could be stored in arrays. In addition, arrays can be used to store data that couldn't be accomodated by parameters or columns. However, because parameters and columns are conceptually simple and natural to work with, they are used in almost all cases. In contrast, the array feature is infrequently used and I won't spend much time on it in this article.

Having introduced the SDDS file protocol, I want to make the definition more formal and introduce some additional terminology. The figure below shows the structure of an SDDS file. The file starts with a header that includes the SDDS version number and a series of definitions of parameters, columns, and arrays. The header in essence defines a data structure, giving the names, data types, and other information about the members of the structure. Following the header are zero or more "data pages," which are simply instances of the data structure.

Returning to the example above of temperatures for a group of cities, one sees that this data would occupy a single page in an SDDS file. One might have additional pages giving data for other groups of cities (in other countries, for example). However, each



page of the file must contain the same parameters and columns. The reason is that there is only one header in each file, and the header defines what data is present.

The reader may wonder why SDDS does not make use of a series of headers and pages within a single file, since this would clearly make it possible to store more general types of data. This is a legitimate question and there is in fact a very good reason that SDDS has this restriction. Lifting this restriction would make SDDS files so flexible that using the operator approach would become quite difficult, both in terms of developing tools and using them. Such files would be potentially so complex that instead of writing generic tools to work with the files, we'd have to return to the writing application−specific programs that work with application−specific files.

In order to allow the user to deal with complex data that cannot be stored in a single SDDS file, the SDDS Toolkit provides tools for taking selected data from one file and placing it in another file. Each of the files can be processed and manipulated separately prior to sharing data between them. This simplifies some aspects of the user's task by reducing the amount of data that must be thought about at one time.

I mentioned above that the header defines the data types of the elements in the file, as

well as their names. SDDS supports six data types: single− and double−precision floating point numbers ("float" and "double" types), short and long integers ("short" and "long" types), one byte characters ("character" type), and character strings ("string" type). Strings may have arbitrary length. SDDS does not presently support unsigned integer or character types.

Introduction to the SDDS Toolkit

In this section, I want to introduce some of the most−used programs and discuss how they are used to work with SDDS files. I won't discuss full details of command syntax, as these are not important at this stage. The most important thing in using the Toolkit successfully is to understand how the various programs modify data sets. Understanding this is actually more important than learning the details of using the programs, which can always be obtained from the manual. In addition, you can obtain the usage message for any program by running the program without any arguments. (The manual discusses the convention for the notation in the usage messages.)

In order to make these examples more concrete, you should try the commands using the sample data files that are available from our Web site at http://www.aps.anl.gov/asd/oag/oaghome.shtml. Here's a description of each of these files:

1) **cityWeather.sdds** contains the tabular data from the example in the previous section. The file contains three columns: **City** (a string column), **Temperature**, and **Humidity**. The latter two columns give double−precision values. The file does not include the parameter data from the example. That will be generated using SDDS tools in an example below.

2) **tSamp1.sdds** contains a table of beam−position−monitor (BPM) data vs. time from a simulation of the APS PAR storage ring. The time data is called **Time**, whereas the BPM columns have names like **P1P1**, **P1P2**, and so on. There is also a column called **Step** that gives the sample number, a parameter called **TimeStamp** that gives the time when the data collection started as a text string, and another parameter called **StartTime** that gives the same information as a double−precision number. (Note that times are stored as numerical values in the usual UNIX fashion, as seconds since some reference date. On many UNIX systems, this reference date is January 1, 1970, at 00:00:00.)

3) **`tSamp2.sdds`** is essentially the same as **`tSamp1.sdds,`** except the data is split into many pages, rather than being collected together on a single page.

4) par.twi contains a table of Twiss parameters and other information for the PAR storage ring. This file contains a large number of columns and parameters, so I won't try to describe them all here.

5) **apsRings.twi** is similar to **par.twi** but it contains three pages of data, one for each of the three rings at APS.

Note that some of these files have the extension **sdds**, but that this is by no means required. In fact, we usually do not use this extension but rather use various extensions that reflect the kind of data in the file. For example, we typically use **twi** as the extension for Twiss parameter data.

Determining What's in an SDDS File

The first thing one needs to do when confronted with an SDDS file is find out what is stored in it. This can be done using the program **sddsquery**, which provides a listing of the names and properties of the parameters, columns, and arrays in an SDDS file. Basic use of **sddsquery** is very easy:

sddsquery *filename*

(The convention for these examples is that anything in **bold** face must be typed literally. Anything in *italics* represents an item that must be supplied by the user. In this case, the user must supply a filename.)

For a file with many elements, you may wish to use a UNIX pagination utility like **less** or **more** to view the data one page at a time:

sddsquery *filename* **| less**

The printout from sddsquery gives a separate table for parameters, columns, and arrays. Each table gives the name of each element and its data type, along with units and other information.

Viewing the Contents of an SDDS File

There are several ways to view the data in an SDDS file. The program **sddsprintout** is designed to provide formatted printouts of data from SDDS files. It will print parameter and column data only. At minimum, one must tell **sddsprintout** what classes of elements you are interested in. For example, to see a printout showing all the column data from the **cityWeather.sdds** file, one would use the command

sddspintout cityWeather.sdds –column

To see all the parameter data in the file **par.twi**, one could use

sddsprintout par.twi –parameter –width=80

where I've added the **width** option to ensure that the printout width is restricted to 80 columns.

One may also give **sddsprintout** specific lists of columns or parameters to print, or wildcard patterns, as in

sddsprintout par.twi –col=s –col=beta* –col=eta?

which would print data the columns **s**, **betax**, **betay**, **etax**, and **etay**. Note that I've abbreviated the option names. All SDDS Toolkit programs allow the use of any unique abbreviation of an option name. We prefer not to make use of this feature when creating permanent scripts, as a given abbreviation is not guaranteed to be unique in all future program upgrades, but it is a great convenience when working at the commandline.

Other functions of **sddsprintout** are to convert data from SDDS into spreadsheet format. This is discussed below in the section entitled ***Converting SDDS to Other Formats.***

Another program that gives text−based printouts from an SDDS file is **sdds2stream**. The original purpose of this program was to allow streaming ASCII data from an SDDS file

into another application, such as a script or non−SDDS program. It is still used this way, although this isn't really very efficient except for relatively small data sets.**sdds2stream** provides some other convenient functions. For example,

sdds2stream *filename* **–rows**

will tell you how many rows are in each page of the file, while

sdds2stream *filename* **–npages**

will tell you how many pages are in the file.

Finally, a completely different way to view the contexts of an SDDS file is using one of the SDDS editors. There are versions written in Tcl/Tk (by Dariusz Blachowicz) and Java (by Robert Soliday). These programs are not strictly part of the SDDS Toolkit and are distributed separately. However, they can be quite convenient in that they permit viewing data in a spreadsheet−like layout and support editing, deleting, searching, merging, and other operations.

Plotting the Contents of an SDDS File

The program **sddsplot** is the primary means of plotting SDDS data. This program is very versatile and is probably the most complex in the Toolkit. However, basic use of **sddsplot** is trivial. **sddsplot** will plot data from parameters, columns, and arrays.

For example, suppose one wants to plot the beta−functions from**par.twi**. One could use

sddsplot par.twi –column=s,beta*

The result is a plot with two curves, but in the same color. To get a different color, one needs an added option, as in

sddsplot par.twi –col=s,beta* –graphic=line,vary

The **graphic** option accepts a "graphic class" as the first qualifier; in this case it's **line**. One may also give **symbol**, **bar**, **impulse**, **errorbar**, and others. In each case, the qualifer **vary** causes **sddsplot** to change the type of the graphic for each plotted quantity.

One may also explicitly specify the graphic type, using the **type** qualifier, as in

**sddsplot par.twi –col=s,betax –graph=line,type=0 **
–col=s,betay –graph=line,type=2

In this example, there are two **column** options. In **sddsplot** jargon, each of these options starts a "plot request." A plot request starts with one or more consecutive options that specify what to plot. It includes zero or more filenames and zero or more other options, which apply only to that request. Any filename or option that preceeds all the plot requests applies to all the plot requests. So the filename**par.twi** in the last example applies to both plot requests. However, the option **–graph=line,type=0** applies only to the first request, while **–graph=line,type=2** applies only to the second request. The concept of plot requests allows using **sddsplot** to compose very complex plots.

Note that the following two commands are very different:

sddsplot –col=s,betax –col=s,betay par.twi

sddsplot –col=s,betax par.twi –col=s,betay

The first of these commands contains a single plot request, because the **column** options are consecutive. Hence, the pairs **(s, betax)** and **(s, betay)** are drawn from **par.twi.** The second command, in contrast, contains two plot requests, the second of which is invalid because it contains no data files.

This may seem unnecessarily complex given that we have only one data file. However, **sddsplot** allows plotting data arbitrary number of data files together. Without the notion of plot requests, it would be impossible to specify how to plot data from many files. There will be examples of using multiple files later in this article.

Like most complex SDDS programs, **sddsplot** has many options, a subset of which are commonly used. Two of these (**column** and **graphic**) have been introduced already. Here's a listing of some of the others:

1. **–device=***deviceType* and **–output=***filename* can be used to change the type of graphical output and direct it to a file. These options should preceed any plot requests. For example, for color postscript output to a file, one can use a command like
   sddsplot –device=cpost –output=*filename***.ps **
   –column=s,beta* par.twi–graph=line,vary
   To get a listing of the available device types, use the command
   sddsplot –listDevices
   One may omit the **output** option and simply pipe the output to a printer, as in
   sddsplot –device=post –col=s,betax par.twi | lpr

2. **–legend** allows requesting legends on the plot. This option may be given prior to any plot requests, in which case all data pairs have legends, or within a plot request, in which case only the data pairs belonging to specific plot requests will have legends. For example,
   sddsplot –legend –graph=line,vary par.twi –col=s,beta*
   will give legends for **betax** and **betay**, whereas
   **sddsplot –graph=line,vary par.twi **
   –col=s,betax –legend –col=s,betay
   will give a legend only for **betax**. You may also specify the legends explicitly, as in
   **sddsplot –graph=line,vary par.twi **
   **–col=s,betax "–legend=spec=Horizontal Beta" **
   –col=s,betay "–legend=spec=Vertical Beta"
   This is a good illlustration of the usefulness of multiple plot requests.

3. **–scales=***xmin,xmax,ymin,ymax* option allows one to specify the x and/or y limits for the plot. For example,
   sddsplot –col=s,betax par.twi –scale=0,10,0,0
   would plot from 0 to 10 in x but with autoscaling in y (because *ymin* and *ymax* are the same). This option is a little primitive. Other related options that can be found in the manual are **–range** and **–limit**.

4. m the different rings with lines as if the pages all pertained to the same ring. A slightly

better plot is obtained with

sddsploThe **–layout=***nx,ny* option allows plotting several panels on a single sheet of paper (or in a single graphics window). The parameters *nx* and *ny* specify the number of horizontal and vertical panels in a grid. Unlike most other **sddsplot** options, the **–layout** option may only be given once and must be given before any plot requests. As an example of its use, consider the following command to plot two quantities in separate panels, one below the other

**sddsplot –layout=1,2 tSamp1.sdds **

–col=Time,P1P1 –end –col=Time,P2P1

The **–end** option is one way to tell **sddsplot** to end a panel. In this case, the initial panel is ended with the first plot request. The second plot request therefore starts on the second panel. As a variation on this, consider

**sddsplot –lay=2,2 –graph=line,vary tSamp1.sdds **

**–col=Time,P1P? –end –col=Time,P2P? –end **

–col=Time,P3P? –end –col=Time,P3P?

5. Another way to separate data onto panels is to use the **–separate** option, as in

   sddsplot –layout=1,4 tSamp1.sdds –separate –col=Time,P1P?

   This is the simplest form of the **separate** option. Another basic form is **–separate=***number*, where *number* gives the number of data pairs to put on each panel. Much more complex operations are possible by combining this option and some of its other qualifiers with the **–groupby** option, which allows sorting and grouping data pairs. Some of these operations will be discussed in later sections.

6. Unlike most other SDDS programs, **sddsplot** by default treats a file as a single data set. Most other programs treat each page of the file as a data set. As a result, one sometimes needs to use the **–split=pages** option to separate data page–by–page. For example, the plot made by the following command is not very useful:

   sddsplot –col=s,betax apsRings.twi

   The data from each page corresponds to one of the APS storage rings. However, the plot shows all the data on one panel and indeed connects data from the different rings with lines as if the pages all pertained to the same ring. A slightly better plot is obtained with

   sddsploThe **–layout=***nx,ny* option allows plotting several panels on a single sheet of paper (or in a single graphics window). The parameters *nx* and *ny* specify the number of horizontal and vertical panels in a grid. Unlike most other **sddsplot** options, the **–layout** option may only be given once and must be given before any plot requests. As an example of its use, consider the following command to plot two quantities in separate panels, one below the other

   **sddsplot –layout=1,2 tSamp1.sdds **

   –col=Time,P1P1 –end –col=Time,P2P1

   The **–end** option is one way to tell **sddsplot** to end a panel. In this case, the initial panel is ended with the first plot request. The second plot request therefore starts on the second panel. As a variation on this, consider

   **sddsplot –lay=2,2 –graph=line,vary tSamp1.sdds **

**–col=Time,P1P? –end –col=Time,P2P? –end **
–col=Time,P3P? –end –col=Time,P3P?

7. **sddsplot –col=s,betax apsRings.twi **
   –graph=line,vary –split=page
   One c**an** clearly see that there are three rings now, but it still isn't very useful as the size of the rings is so different. Adding another option gives a useful result:
   **sddsplot –col=s,betax apsRings.twi **
   –split=page –separate
   where I've removed the **–graphic** option as it isn't necessary when plotting one item per panel. In this command, the pages are split into individual data sets, which are plotted on separate panels.
   **sddsplot –col=s,betax apsRings.twi **
   –graph=line,vary –split=page
   One can clearly see that there are three rings now, but it still isn't very useful as the size of the rings is so different. Adding another option gives a useful result:
   **sddsplot –col=s,betax apsRings.twi **
   –split=page –separate
   where I've removed the **–graphic** option as it isn't necessary when plotting one item per panel. In this command, the pages are split into individual data sets, which are plotted on separate panels.

8. In several of the plots shown above of data from **tSamp1.sdds**, the time axis labels show large numbers that don't have any clear meaning. These are the UNIX time−stamp values, giving seconds since January 1, 1970. A more useful display can be obtained using the **–ticks=xtime** option, which instructs **sddsplot** to make time–style ticks for the x axis. For example, consider
   sddsplot –col=Time,P1P1 –ticks=xtime tSampl.sdds
   In this plot, the x axis label is replaced by a string giving the starting time of the plot, while the x tick labels show fine−scale time information, in this case minutes and seconds information. **–ticks=xtime** can be used for time scales from fractions of a second to decades.

9. The badly−named **–mode** option is used to perform a number of special transformations of the data prior to plotting. For example, it can be used to take the base−ten logarithm of the data, to normalize the data, and to apply automatically−computed offsets. Plotting data using a log−scale in**sddsplot** involves specifying two actions: taking the logarithm and (optionally) using special log scales, as in
   sddsplot –col=s,betay par.twi –mode=y=log,y=special
   Another way to do this is to combine the **–mode** and **–ticks** options, as in
   sddsplot –col=s,betay par.twi –mode=y=log –ticks=ylog
   Using the **–ticks** option allows accessing other features such as non−exponential labels on the ticks.

10. Another frequently−needed plotting feature is to use different scales for different data pairs. For example, one might want to plot **betax** and **betay** on one scale and **etax** on

another. This can be done using the **–yscales** option, as in

```
sddsplot par.twi –graph=line,vary \
  –col=s,beta? –yscales=id=beta \
  –col=s,etax –yscales=id=eta
```

In this command, a **–yscales** option is given for each plot request. In the first instance, the command specifies use of a new y scale with identifier ("id") "beta," whereas in the second instance, a new y scale is used with identifier "eta." The use of identifiers for the scales allows the user to create scales and use them for specific data pairs. For example, the above command could be rewritten and improved as follows

```
sddsplot par.twi –graph=line,vary \
  –col=s,betax –yscales=id=beta "–legend=spec=Beta x" \
  –col=s,betay –yscales=id=beta "–legend=spec=Beta y" \
  –col=s,etax –yscales=id=eta "–legend=spec=Eta x"
```

The y scale "beta" is referred to in two plot requests now. The advantage of doing this in this example is that it allows specifying additional options for each plot request.

Using sddsprocess for Basic Analysis and Filtering

The program **sddsprocess** is, like **sddsplot**, very powerful and potentially complicated. Like **sddsplot**, basic use of **sddsprocess** isn't hard. **sddsprocess** provides the following functions, among others: evaluating equations to create new columns and parameters in a file; filtering data based on values in columns and parameters; and computing statistics of column data to create parameters.

Equation Evaluation

sddsprocess uses RPN (Reverse Polish Notation) for specifying equations. These equations may contain the names of parameters and columns along with mathematical operators of various types. This facility is based on the RPN interpreter used by the programs **rpn** and **rpnl**, both of which are distributed with the SDDS Toolkit. One way to gain familiarity with the operators available is thus to run **rpn** and give the **help** command. It is also documented in the SDDS Toolkit manual.

Here's an example of using the equation feature in **sddsprocess**. In this example, I define a new parameter giving the x emittance of the PAR storage ring, then use that parameter and the x beta function to compute the monoenergetic beam size, given by the equation $\sigma = \sqrt{\epsilon * \beta}$

```
sddsprocess par.twi par1.twi \
  −define=parameter,emitx,3.6e−7,units=m \
  −define=column,Sx,"emitx betax * sqrt",units=m
sddsplot −column=s,Sx par1.twi
```

Statistics and Other Column Processing

sddsprocess provides for computation of various statistics of column data, with the results of the computations being placed in newly−created parameters. The command syntax permits wildcards in the names of the columns to be processed, allowing easy

processing of large numbers of columns with a single command. Types of processing include average, rms, standard deviation, minimum, maximum, mode, median, quartile and decile ranges, percentiles, and so on. A full list along with details of syntax is available from the manual.

We can use **sddsprocess** with the file **cityWeather.sdds** to compure some average temperatures and find the most humid city. This is done with the following command:

```
sddsprocess cityWeather.sdds cityWeather.proc \
  -process=Humidity,ave,Average%s \
  -process=Temp*,ave,Average%s \
  -process=Humidity,max,MaximumHumidity \
-process=Humidity,max,MostHumidCity,position,functionOf=City
```

This command contains four **-process** options. The first of these finds the average of the data in the **Humidity** column and puts the result in a new parameter called **AverageHumidity**. The syntax "**Average%s**" tells sddsprocess to compose the new parameter name by substituting the column name for %s. This syntax is needed when using wildcards, and it saves typing even when not using wildcards. The second option finds the average of all columns matching the wildcard sequence "**Temp***". Of course, there is only one such column in our file. In this case, the syntax saves typing out the entire word. The third option simply finds the maximum value of the **Humidity** column. The final option is more complex, because it involves finding the value in the column **City** corresponding to the maximum value in the column **Humidity**. To perform such two-column operations, the **-process** option requires that the first column (**Humidity**) be declared a function of another column (**City**). In addition, for this particular case, one must specify that one wants the **position** of the maximum, i.e., the value of **City** rather than the value of **Humidity.**

We'll see some more examples of using the **-process** option a little later.

Filtering Data

Another feature of **sddsprocess** is the ability to filter data based on the contents of columns or parameters. This is accomplished using the **-filter**, **-match**, and **-test** options. When the filtering conditions involve numbers, **-filter** is generally used, although **-test** is also sometimes employed. When the filtering uses string or character data, **-match** is used.

Here are some simple examples of using the **-filter** and **-match** options. Suppose you wanted to find the average beta functions at the end of all the quadrupoles in **par.twi**. These elements have **ElementType** of "QUAD". The command would be

```
sddsprocess par.twi -pipe=out \
 -match=column,ElementType=QUAD \
 -process=beta?,average,%sAve \
 | sddsprintout -pipe -parameter=beta?Ave
```

Similarly, if you wanted to find the average beta functions for all elements in the first 10m of the ring that are also quadrupoles, you could use the command

```
sddsprocess par.twi −pipe=out \
 −match=column,ElementType=QUAD \
 −filter=column,s,0,10 \
 −process=beta?,average,%sAve \
| sddsprintout −pipe −parameter=beta?Ave
```

Note that the order of items on the **sddsprocess** commandline is important. The following command finds the average beta functions for *all* elements, after which it removes all rows that do not pertain to QUAD elements, and finally computes the average for all quadrupoles:

```
sddsprocess par.twi −pipe=out \
 −process=beta?,average,%sAllAve \
 −match=column,ElementType=QUAD \
 −process=beta?,average,%sQuadAve \
| sddsprintout −pipe −parameter=beta?*Ave
```

This feature of **sddsprocess** allows combining many processing steps on a single commandline. The steps are implemented for each page in the order that they are given on the commandline. One limitation is that wildcard sequences may not refer to elements created by previous steps. All wildcard sequences are resolved with reference to the elements in the input file.

Converting SDDS to Other Formats

Sometimes it is convenient to convert SDDS data into other formats. For example, you may want to import the data into a program that is not SDDS−compliant. There are good ways and bad ways to do this. A commonly−used practise that I *very strongly discourage* is to use the program **sddsconvert** with the **−ascii** option, then edit out the SDDS header to produce a text file. The main reason this is bad is that the definition of SDDS provides no guarantee of the order of the data in the file. For example, whether the first column of data is **betax** or **ElementName** is irrelevant in the SDDS context and hence is not specified. If you want to ensure that your conversion gives the same results every time, you must use another utility for this purpose. The second reason this is a bad method is that it is tedious and error−prone. SDDS provides systematic, repeatable methods of emitting non−SDDS data, which I encourage you to use.

There are several utilities to choose from. **sdds2stream** is a good choice for single page data files or cases where you don't need any labels on the data. You must explicitly name each parameter or column that you want in the output, which guarantees that there are no surprises. Another popular choice is **sddsprintout**, which can produce spreadsheet−ready output. Note that this program will accept wildcards in the names of the columns, so there is a potential pitfall in that your output may be different if unexpected, matching columns or parameters are present in the file. This is of particular concern if the **sddsprintout** command will be embedded in a script that may be used over and over again. Here's how to use **sddsprintout** to make comma−separated−value spreadsheet data:

sddsprintout –spreadsheet=csv –column=* cityWeather.sdds cityWeather.csv

This command creates the file **cityWeather.csv**, which can be imported directly into most spreadsheet programs.

An easier−to−use program than **sddsprintout** is **sdds2plaindata**. This program will provide plain data in either ASCII or binary formats. The "plain data" format has helpful information such as the number rows in a page. It will not, however, provide a spreadsheet−ready file (with column headers) such as generated by **sddsprintout**.

Converting Non−SDDS Data to SDDS

There are a number of programs that allow converting non−SDDS data to SDDS. Among these are **csv2sdds** and **plaindata2sdds**. These programs perform pretty much the same function, but with some differences. **plaindata2sdds** is the more general program, but **csv2sdds** is easier if you happen to have comma−separated−value data. For example to convert the cityWeather.csv file back to SDDS, one could use the following command:

**csv2sdds cityWeather.csv cityWeather.sdds1 –skiplines=1 **
**–column=name=City,type=string –column=name=Temperature,type=double **
–column=name=Humidity,type=double

(The **–skiplines** option is used to force **csv2sdds** to ignore the first line of the file, which contains the column headers.)

"Hierachical" Data Processing

The SDDS toolkit excels at what I call "hierarchical" data processing. This means processing the results of prior processing. It usually involves using the program **sddscollapse**, often in combination with **sddsprocess**. The purpose of the latter program is to simplify a data set by removing all of the column data and turning the parameters into columns. If the columns were previously analyzed using **sddsprocess**, using **sddscollapse** produces a new data set that contains the results of the analysis, with each row of the new data set corresponding to a page of the original data set.

Let's see how this works with the file tSamp2.sdds, which contains many pages of data. Suppose you wanted to compute the average value (over pages) of the standard deviation (in a page) for each of the P?P? columns in this file. Here's how you'd do it:

**sddsprocess tSamp2.sdds –pipe=out **
**–process=P?P?,standardDeviation,%sStDev **
**| sddscollapse –pipe **
**| sddsprocess –pipe **
**–process=P?P?StDev,max,%sMax **
**–process=P?P?StDev,ave,%sAve **
**| tee tSamp2.proc1 **
| sddsprintout –pipe –parameter=P?P?StDev???

The first step in this command processes each of the columns P?P? to find the standard

deviation of the values for each page. In the second step, **sddscollapse** is used to throw out the column information and convert the parameters into columns. Each *row* in the output of **sddscollapse** corresponds to a *page* in the input. Hence, we can now use the **–process** option of **sddsprocess** to work on the standard deviations. In this case, we find the maxima and averages of the standard deviations across pages. Note that we are always performing operations independently for data pertaining to each of the original columns.

Notice the use of the UNIX **tee** command to make a copy of the data before piping it into **sddsprintout**. We can use this copy for another step of analysis. Suppose that we now want to plot the average and maximum standard deviations as a function of the name of the originating column. We know that the parameters we are interested in have names like P1P1StDevAve, P1P1StDevMax, etc. If we use **sddscollapse** again, we can convert these parameters into columns. The file at this point has only a single data page and a single row in that page, and contains two columns for each column of the original file. We'd like to rearrange the data so that we have a column containing the name of the original column, a column of averages of standard deviations, and a column of maxima of standard deviations. Here's how to do it:

```
sddscollapse tSamp2.proc1 –pipe=out \
| sddscollect –pipe=in tSamp2.proc2 \
–collect=suffix=StDevAve \
–collect=suffix=StDevMax
```

```
sddsplot –column=Rootname,StDev??? –graph=line,vary  \
 –legend tSamp2.proc2
```

We've now performed processing of the original data through two levels. I.e, we've processed the processing. We can keep doing this. For example, suppose you wanted to know the median and maximum of the "StDevMax" values, as well as the name of the original column for which the StDevMax was greatest. You'd find these as follows:

```
sddsprocess tSamp2.proc2 –pipe=out \
–process=StDevMax,median,%sMedian –process=StDevMax,max,%sMax \
–process=StDevMax,max,%sMaxSigName,func=Rootname,posi \
| sddsprintout –pipe –parameter=StDevMax*
```

It is hard to do much more processing with this dataset, but imagine that we had a number of data sets of the same type. We could process each of them as above, then combine them to continue the processing. For example, we might want to find out which was the noisiest signal for each dataset, or plot the value of that signal against some parameter that varies between datasets (e.g., the time of acquisition). The combining of datasets is accomplished using the program **sddscombine.**

I hope it is apparent from this discussion that SDDS can be used to perform very complex, multi−level processing and reduction of data. This kind of data processing is frequently used in dealing with data from experiments at APS. Most often, the

processing is performed by scripts, as it is easier to keep track of the many levels of analysis that way. The "for" and "foreach" looping commands that are found in script languages like csh, bash, and tcl are very useful in this context.