BURT:
Back Up and Restore Tool

Nicholas T. Karonis
Argonne National Laboratory
Advanced Photon Source
Accelerator Systems Division/Controls Group
November 1994

Table of Contents

Chapter 1: Introduction

In this document we address the problem of backing up and restoring sets of values in databases whose values are continuously changing. In doing so, we present the Back Up and Restore Tool (BURT). In this presentation we provide a theoretical framework that defines the problem and lays the foundation for its solution. BURT is a tool designed and implemented with respect to that theoretical framework. It is not necessary for users of BURT to have an understanding of that framework. It was included in this document only for the purpose of completeness.

BURT's basic purpose is to back up sets of values so that they can be later restored. Each time a back up is requested, a new ASCII file is generated. Further, the data values are stored as ASCII strings and therefore not compressed. Both of these facts conspire against BURT as a candidate for an archiver. Users who need an archiver should use a different tool, the Archiver[2].

1. Assumptions

BURT is just one of the tools in the Experimental Physics Industrial Control System (EPICS). It is assumed that the reader is familiar with EPICS and all its associated terminology, e.g., databases, Input/Output Controllers (IOCs), process variables, etc. A full description of EPICS can be found in [6].

BURT allows the user to make use of a number of facilities provided by the C programming language and the Standard C Library. Users of BURT are not required to use these features, however they are nonetheless available. These features are the **#define** and **#include** directives, the syntax of boolean expressions found in C, and functions found in the Standard C Library, e.g., **strcmp()**. It is assumed that those users who wish to take advantage of these features are familiar with them, and hence, we provide no description of them in this document. A full description of these topics in C can be found in [5] and [7].

2. About This Manual

In the next chapter we provide a general overview of BURT, i.e., we present the suite of programs that comprise BURT. Following that (Chapter 3) is a detailed description of all the components and terminology found in BURT, e.g., request files, snapshot files, etc. Chapters 4 and 5 describe how to execute BURT from a UNIX prompt and its Graphic User Interface (GUI), respectively. Chapter 6 discusses advanced features in BURT, e.g., generating your own SDDS snapshot files. Finally, we conclude with Chapter 7 where we present the theoretical framework that BURT is based upon and a discussion of where and why BURT, as implemented, falls short of that framework.

BURT is also capable of understanding Self Describing Data Set (SDDS) files as well as interacting with DevLib for the backing up and restoration of devices, although users of BURT are not *required* to use or even be familiar with either. We provide no description of SDDS or DevLib, instead we refer the reader to [1] for SDDS and [8] for DevLib.

Throughout the manual items surrounded by braces, i.e., {...}, should be interpreted as optional. These are found in the description of the syntax of files and as options the user may specify when executing BURT from a UNIX prompt. Filenames appear in italics. UNIX commands are prefaced with a UNIX prompt **>** and all commands, command line switches, and function names appear in boldface.

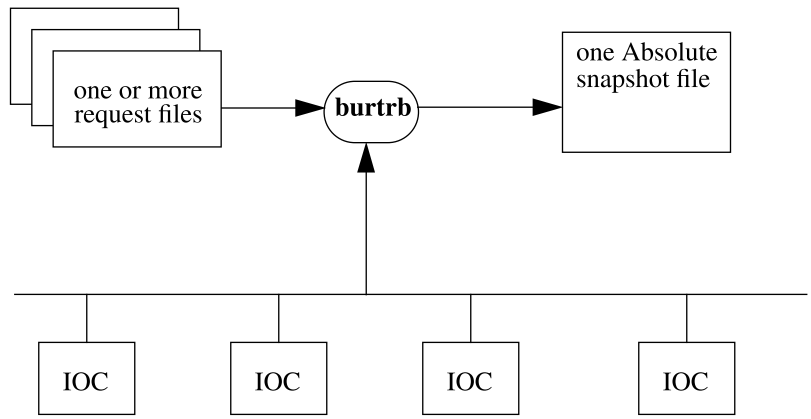
Chapter 2: BURT Overview

This chapter provides an overview of the five programs that comprise BURT. The details about the components that appear in this chapter (snapshot files, request files, dependency files, RO and RON process variables and devices) are described in Chapter 3. Again, this chapter merely provides a general overview.

The first two programs, **burtrb** and **burtwb**, are used to backup and restore EPICS process variable and device values from and to databases that reside on IOCs, respectively. The next two programs in the BURT suite of programs, **burtmath** and **burtset**, are used to modify and combine snapshot files and request files, respectively.

1. burtrb

The backup program, **burtrb**, takes a set of one or more ASCII files called request files, (request files are fully described in Chapter 3 Section 1). They identify the process variables and devices to backup. **burtrb** makes a list of all the process variables and devices from the set of request files and takes a snapshot of their values from the IOCs. Their names and values are written to a single snapshot file. Snapshot files are typed (snapshot files and their types are described in Chapter 3 Section 2). The type of snapshot file **burtrb** always creates an Absolute snapshot file.

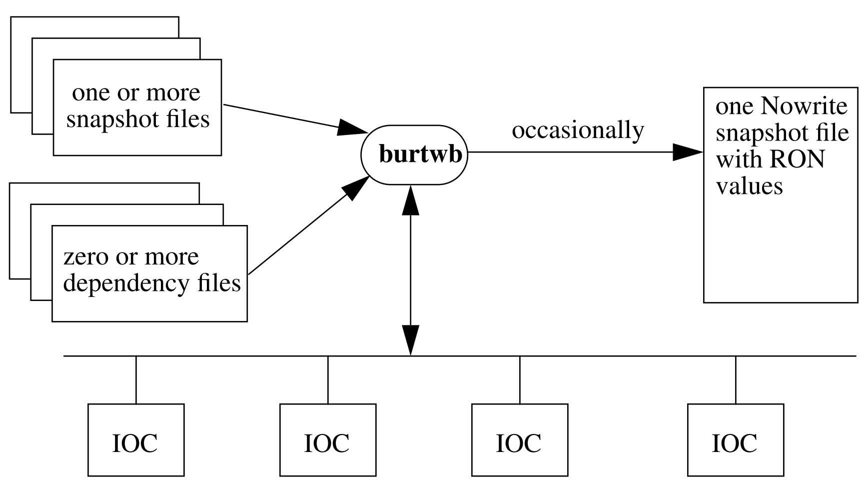


2. burtwb

The restore program, **burtwb**, takes a set of one or more ASCII files called snapshot files, for example like those generated by **burtrb**, and sets the values of the process variables and devices on the IOCs accordingly.

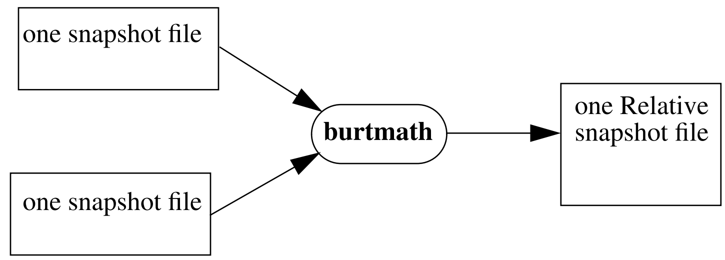
burtwb restores all the values either conditionally or unconditionally. The user can specifying a set of conditions that **burtwb** will test before restoring *any* of the values. The conditions are supplied to **burtwb** in the form of user-defined dependency files, (described in Chapter 3 Section 3). Dependency files specify conditions about values on the IOCs and the snapshot files. Depending on the specified conditions, the values on the IOCs, and the values in the snapshot files, **burtwb** restores either all or none of the values to the IOCs. If the user does not specify any dependency files, then **burtwb** restores the values unconditionally.

In either case, conditional or unconditional restoration, any Read Only Notify (RON, described in section Chapter 3 Section 1.2) values found in any of the snapshot files are reported in a Nowrite snapshot file. If there are no RON values, then no snapshot file is generated.

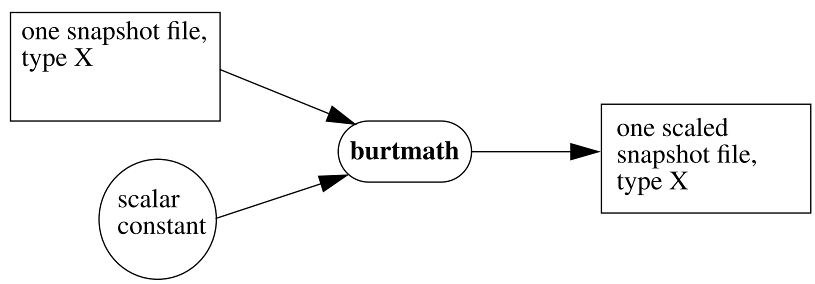


3. burtmath

The arithmetic program, **burtmath**, operates on snapshot files. It allows the user to either add or subtract corresponding values found in two snapshot files. As output it produces a Relative snapshot file containing the sum or difference of the two input snapshot files.

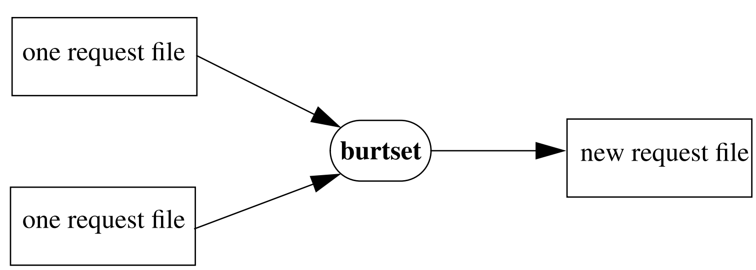


burtmath also allows the user to multiply the contents of a single snapshot file by a scalar constant. The output is a snapshot file that inherits its type from the input snapshot file.



4. burtset

The set operation program, **burtset**, operates on request files. It allows the user to perform mathematic set operations, (union, intersection, or difference), on two request files to produce a new request file.



5. burtconvertsnap

The final BURT program, **burtconvertsnap**, is a conversion program. It operates on a single snapshot file by converting its format to either SDDS or non-SDDS.

6. Execution

BURT can either be executed directly from a UNIX prompt or from a Graphic User Interface (GUI) that we have provided. The instructions regarding how to execute BURT in either mode (UNIX or GUI) as well as a detailed description of BURT's components are in the following chapter.

Chapter 3: BURT Components

In this section we provide detailed descriptions of all of BURT's components, e.g., request files, snapshot files, etc., as well as descriptions of how BURT uses each of these components.

1. Request Files

Request files are ASCII files that identify a list of process variables and devices the user wishes to backup. They are used as input to BURT's backup program, **burtrb**, to produce snapshot files and its set operation program, **burtset**, to produce other request files.

It is not possible to distinguish between atomic and composite devices in request files. However, it is important to know that RO and RON tags as well as BackupMsg and RestoreMsg messages are inherited by all the elements in a composite device.

1.1. Request File Formats

BURT recognizes request files in two formats, non-SDDS and SDDS. Both of the BURT programs that accept request files as input allow the user to intermix the formats freely in any single execution. These two formats are not equivalent, syntactically nor functionally. In short, non-SDDS request files allow the user to use the C **#include** and **#define** directives while SDDS request files do not. On the other hand, the SDDS request files allow the user to specify devices while non-SDDS do not.

1.2. Read Only (RO) and Read Only Notify (RON) Tags

When composing a request file, the user has the option of tagging each item (process variable or device) as Read Only (RO), Read Only Notify (RON), or not tagging it at all. The tags are propagated to the snapshot file by **burtrb**. They instruct **burtrb** to backup the values and place them into the snapshot file.

When the restore program **burtwb** is supplied that snapshot file as input, those items that were tagged RO or RON do *not* have their values written back to the IOCs. Further, if there are any RON tags in the snapshot file, **burtwb** creates a Nowrite snapshot file (snapshot files and their types are described in Section 2) and places all the RON values into that file. The values of those items that were not tagged are restored to the IOCs.

1.3. non-SDDS Request Files

Non-SDDS request files are ASCII files that identify a list of process variables the user wishes to backup. It is not possible to specify a device in a non-SDDS request file, only process variables.

BURT ignores blank lines and lines that begin with % in non-SDDS files. This allows the user to augment non-SDDS files with comment lines that begin with %.

Additionally, BURT processes these files with the C preprocessor. This allows the user to take advantage of the **#define** and **#include** directives provided by the C programming language. This is illustrated in the example at the end of this section.

The rest of the file are lines that have the following format:

{RO or RON} pvarname {nelements}

where

RO or RON - Optional. Read Only or Read Only Notify. This tags the process variable as either Read Only or Read Only Notify. It instructs **burtrb** to backup the process variable and place its value into the snapshot file. However, when the snapshot file is used as input to BURT's restore program **burtwb**, this value is not restored to the IOC.

pvarname - The record name and field name of the process variable to read. The default field name is VAL.

nelements - Optional. Number of elements. For those process variables that contain more that one data value, e.g., vectors, nelements can be used to backup the first n elements. The default is to backup the all the data values of the process variable.

1.4. Non-SDDS Request File Example

Following is an example of non-SDDS request files and how BURT processes them. Consider the following three request files; *req1*, *req2*, and *req3*.

req1

```
%
% Example request file: req1
%

#define PREFIX LINAC
#include "req2"

 PREFIX:rec1
```

req2

```
%
% Example request file: req2
%

RO PREFIX:rec2
```

req3

```
%
% Example request file: req3
```

```
%
PREFIX:rec3
LINAC:rec4 5
RON LINAC:rec5
```

Here we examine how **burtrb** and **burtset** would process these files to construct a single list of process variables if they were only explicitly given *req1* and *req3* as input.

The request files are processed in the order in which they are specified. Assume that *req1* was specified first (the syntax for BURT commands is in Chapter 4). It is processed by the C preprocessor which notes the **#define** for `PREFIX` and then includes the second request file *req2*. That request file is also processed through the C preprocessor, but does not find any **#define** or **#include** directives. Processing continues and the first process variable request is encountered, `RO PREFIX:rec2` (from *req2*). The **#define** from *req1* is used and the request is changed to `RO LINAC:rec2`. This completes the processing of *req2* and the processing of *req1* continues. The next request is translated and becomes `LINAC:rec1`. Note that the **#define** found in request file *req1* was applied to the request file *req2* because it appeared *before* the **#include** that brought *req2* into *req1*. This concludes the processing of *req1*.

Now the second request file, *req3* is processed. It is also processed by the C preprocessor without any effect. Note the **#define** for `PREFIX` worked in *req1* and *req2*, but not *req3*. The final list after processing the *req1* and *req3* is:

```
RO LINAC:rec2
LINAC:rec1
PREFIX:rec3
LINAC:rec4 5
RON LINAC:rec5
```

This is the list of process variables that **burtrb** will attempt to backup.

1.5. SDDS Request Files

SDDS request files are ASCII files that identify a list of process variables and devices the user wishes to backup. Unlike non-SDDS request files, BURT does not process SDDS request files through the C preprocessor. This means that users cannot take advantage of the **#define** and **#include** directives provided by the C programming language when composing SDDS request files. However, using SDDS request files enables the user to specify devices, something not possible in non-SDDS request files.

BURT requires two columns in SDDS request files (ControlName and ControlType) and looks for four other optional columns. BURT ignores all other information (columns and parameters) in the SDDS file. In short, what makes an SDDS file a valid request file is the presence of the two required SDDS_STRING columns, ControlName and ControlType. This point is illustrated further in Section 2.3 where we discuss SDDS Snapshot Files. The table below describes the information BURT is interested in when processing SDDS request files.

Column name	SDDS Type	Required/ Optional	Contents	Default Value	Default Meaning
ControlName	SDDS_STRING	required	PV or Device Name		
ControlType	SDDS_STRING	required	“pv” or “dev”		
BackupMsg	SDDS_STRING	optional	Dev read msg. must be “-” for pv	“-”	“read”
RestoreMsg	SDDS_STRING	optional	Dev write msg. must be “-” for pv	“-”	“set”
Count	SDDS_LONG	optional	number of elements. must be 0 for devices	0	native count

Column name	SDDS Type	Required/ Optional	Contents	Default Value	Default Meaning
ControlMode	SDDS_STRING	optional	“-”, “RO”, or “RON”	“-”	Restore value to IOC

The columns BackupMsg and RestoreMsg are used for devices only while the column Count is used for process variables only.

If an optional column is missing, the default value is used and is interpreted according to the default meaning. If an optional column is supplied, the user may supply the default value. For example, the user is *required* to supply the default value “-” when supplying the BackupMsg column for a row specifying a process variable (ControlType = “pv”). In either case, whether the default value is implied by the omission of an optional column or it is explicitly provided, BURT interprets each default value accordingly. For example, the default value for the column BackupMsg is “-”. That value is interpreted by BURT to mean the message “read”.

1.6. SDDS Request File Examples

These points are illustrated in the following two example request files; *req4* and *req5*.

req4

```
SDDS1
&column name=“ControlName”, type=string &end
&column name=“ControlType”, type=string &end
&data mode=ascii, no_row_counts=1 &end
LINAC:dev1 dev
LINAC:rec1 pv
```

Req4 is an SDDS request file with only the required columns. It specifies one process variable and one device. Neither are tagged RO or RON, and in both cases the native count will be backed up. Consider *req5*, a more robust request file.

req5

```
SDDS1
&column name=“ControlName”, type=string &end
&column name=“ControlType”, type=string &end
&column name=“Count”, type=long &end
&column name=“ControlMode”, type=string &end
&column name=“BackupMsg”, type=string &end
&column name=“RestoreMsg”, type=string &end
&data mode=ascii, no_row_counts=1 &end
LINAC:rec1 pv 0 RO - -
SR:dev1 dev 0 - - -
LINAC:rec2 pv 5 - - -
SR:dev2 dev 0 RO read set
SR:dev3 dev 0 RON get put
```

Here we have two process variables and three devices. The first process variable, `LINAC:rec1`, is tagged RO and requests the native count. The second process variable, `LINAC:rec2`, is not tagged and requests the first five elements of its value, presumably a vector. Note that both of the process variables specify “-” as their BackupMsg and RestoreMsg. In SDDS request files that have these columns, all process variables must specify “-” in those columns.

Following the process variables are devices. Note that the three devices all have a Count of 0. Like BackupMsg and RestoreMsg for process variables, all devices must specify a Count of 0. The first device, `SR:dev1`, is not tagged and implicitly specifies the default BackupMsg “read” and RestoreMsg “write”. The second device, `SR:dev2`, is tagged RO and explicitly specifies the default BackupMsg ‘‘read’’ and RestoreMsg ‘‘set’’. The third device, `SR:dev3`, is tagged RON and specifies its own BackupMsg and RestoreMsg, ‘‘get’’ and ‘‘put’’, respectively.

2. Snapshot Files

Snapshot files are ASCII files that contain process variables and their values, presumably to be restored to IOCs. They are used as input to BURT’s restore program **burtwb**, its arithmetic program **burtmath**, and its snapshot conversion program **burtconvertsnap**.

Typically, snapshot files are generated as output of BURT programs, e.g., **burtrb**, **burtwb**, and **burtmath**. As such, those BURT programs that accept them as input (**burtwb**, **burtmath**, and **burtconvertsnap**) expect and insist that they conform to a very rigid format. For this reason, ***users should never edit snapshot files. Those who do, do so at their own risk***. In Chapter 6 we describe the advanced features of BURT. There we describe that rigid format so that users can create their own snapshot files from their own programs.

BURT recognizes snapshot files in two formats, non-SDDS and SDDS. All of the BURT programs that accept snapshot files as input allow the user to intermix the formats freely in any single execution. Like non-SDDS and SDDS request files, the two formats of the snapshot files are not equivalent. Aside from their syntactic differences, SDDS snapshot files may contain devices as well as process variables while non-SDDS snapshot files may only contain process variables. This is not to say that if a request file has devices it cannot produce a non-SDDS snapshot file. If the user requests that a non-SDDS snapshot be produced, BURT resolves all device names to their process variable constituent(s) and places the process variable names into the non-SDDS snapshot file. For SDDS snapshot files composite devices are reduced to their atomic device constituents and they are placed into the snapshot file.

Snapshot files are ASCII files that start with a header section and conclude with the snapshot data. The contents found in the header section of both non-SDDS and SDDS snapshot files are identical. They only differ syntactically. The header identifies who generated the snapshot and when. It also contains any keywords and/or comments the user saw fit to add at the time the snapshot was generated. Finally the header identifies what type of snapshot file it is; Absolute, Relative, or Nowrite.

2.1. Types of Snapshot Files

BURT supports three types of snapshot files: Absolute, Relative, or Nowrite snapshot file. The type of snapshot file instructs BURT (specifically **burtwb**) how to restore all the values in that snapshot file.

When the values in an Absolute snapshot file are applied to the IOCs, they *replace* the values that exist on the IOCs by default. When the values of a Relative snapshot file are restored, they are *added* to the values on the IOCs by default. **Burtwb**’s treatment of each of these types of files can be changed at the user’s discretion, i.e., Absolute written as Relative and vice versa.

The third type of snapshot file is Nowrite. It is a special type of snapshot file typically generated as output from **burtwb**. When the values of a Nowrite snapshot file are applied to the IOCs, it has no effect on them.

2.2. non-SDDS Snapshot Files

Non-SDDS snapshot files are ASCII files. They have a header which is followed by data. The structure of the file is very rigid, i.e., BURT expects that it generated the snapshot file and so it expects everything to be where it put it.

2.3. SDDS Snapshot Files

SDDS snapshot files are ASCII files. They have a header which is followed by data. The header is in the form of SDDS fixed value parameters and SDDS associate files. The names of the request files used to generate an Absolute snapshot file appear as associate files. The following table describes the parameters found in the header.

Parameter Name	SDDS Type	Value
TimeStamp	string	when snapshot was taken
LoginId	string	loginid of user who took snapshot
EffectiveUID	string	effective UID of user who took snapshot
GroupID	string	group ID of user who took snapshot
BurtKeywords	string	user supplied keywords
BurtComments	string	user supplied comments
SnapType	string	type of snapshot

Following the header information is the data. The data appears in columns. The following table describes the columns of the data portion.

Column Name	SDDS Type	Contents
ControlName	SDDS_STRING	PV or Device name
ControlType	SDDS_STRING	type of entity, “pv” or “dev”
Lineage	SDDS_STRING	lineage of composite devices, for devices only, “-” for pvars
BackupMsg	SDDS_STRING	device read message, for devices only, “-” for pvars
RestoreMsg	SDDS_STRING	device write message, for devices only, “-” for pvars
ControlMode	SDDS_STRING	read only tag, “-”, “RO”, or “RON”
Count	SDDS_LONG	number of elements, 0 for devices
ValueString	SDDS_STRING	value

There are two things of interest to note about the data columns. The first is that all snapshot files have the two columns required by request files: ControlName and ControlType. This means that snapshot files are valid request files, i.e., snapshot files can be used as request files. In doing so, BURT uses the columns ControlName and ControlType, the columns required in all request files, as well as the columns BackupMsg, RestoreMsg, ControlMode, and Count. BURT ignores the columns Lineage and ValueString as well as all the parameters found in the header.

The second thing to note about the data columns is that the column ValueString is a character string. This is an explicit example of something that is true throughout all of BURT, for the purposes of back up and restore, BURT views all data values as character strings. Only in BURT’s arithmetic operation program, **burtmath**, does it ever consider values as numeric. This will typically not cause any trouble to the user who simply backs up and restores values, but is something that users should be made aware of if they are going to use snapshot files with tools other than BURT, e.g., Mathematica, PV Wave, or applications they have written themselves.

3. Dependency Files

BURT allows the user to check that certain conditions are true before restoring its values with **burtwb**. The user does this by specifying the conditions in an ASCII file called a dependency file. Dependency files have a special syntax all their own. As such, they do not have the two formats, SDDS and non-SDDS, like the request and snapshot files do. However, like non-SDDS request files, they are processed by the C pre-processor. This enables users to take advantage of the **#define** and **#include** directives found in C. Blank lines and lines beginning with % are ignored.

Currently, conditional restoration is only available with the command line version of BURT and not available through its Graphic User Interface. All non-blank and non-comment lines in a dependency file have the following format:

{**HALT or CONT**} **condition**

where

HALT or CONT - Optional. Failure directive. This directs **burtwb** in the event that this condition is false or cannot be evaluated (see **ca_get()** and **ss_get()** below). HALT notifies the user that the condition failed and stops processing all together. This means that no more conditions are evaluated and none of the snapshots are restored to the IOCs. CONT also notifies the user that the condition failed, but continues processing. If HALT or CONT are omitted, **burtwb** notifies the user that the condition has failed and prompts the user whether or not to continue.

condition - Condition. The condition to be evaluated. BURT conditions follow the same syntax as Boolean expressions (expressions that are either true or false, like in an if statement) in the C language. Some of the relational operators are == (equal), != (not equal), <, <=, >, >=. etc. Some of the Boolean operators and connectors are ! for ‘‘not’’ (negation), && for ‘‘and’’ (conjunction), and || for ‘‘or’’ (disjunction). When comparing character strings the function **strcmp()** should be used.

Note, ***dependency files may only refer to process variables, not devices***.The conditions in the dependency files can refer to the values of process variables on the IOCs as well as values in snapshot files. They do this by making calls to the **ca_get()** and **ss_get()** families of functions, respectively.

3.1. ca_get()

This family of function calls enables a condition to retrieve current values of process variables from IOCs. Each process variable has a native data type, e.g., integer, floating point, etc. When requesting values from the IOCs, the users may retrieve the data in its native form by making the following call:

ca_get(pvarname)

On the other hand, the user may wish to retrieve the data as a type other than the value’s native type. If so, the user may make any of the following calls:

ca_get_string(pvarname)

ca_get_int(pvarname)

ca_get_short(pvarname)

ca_get_float(pvarname)

ca_get_enum(pvarname)

ca_get_char(pvarname)

ca_get_long(pvarname)

ca_get_double(pvarname)

However, in using any of the calls in the **ca_get()** family, the user is restricted in that these function calls return only the first data element of the named process variable. In other words, it works fine for all scalar values but will only retrieve the first value in any vector.

If the process variable cannot be found, the entire condition in which the **ca_get()** call was made is not evaluated. In that case, **burtwb** treats the entire condition as being false and follows the failure directive of the condition as described above.

3.2. ss_get()

This family of function calls enables a condition to retrieve values of process variables from snapshot files. There are only two calls in this family:

ss_get(pvarname)

ss_get_string(pvarname)

ss_get() retrieves the value as it appears in the snapshot file. **ss_get_string()** also retrieves the value as it appears in the snapshot file, and surrounds it with double quotation marks thus turning it into a character string constant with respect to the C language.

These calls will only work on those snapshot values that represent a single data element. If the process variable name is not found in the snapshot files, the entire condition in which the **ss_get()** call was made is not evaluated. In that case, **burtwb** treats the entire condition as being false and follows the failure directive of the condition as described above.

Consider the following example. Recall the request files *req1-req3* from Section 1.3 above. They were used to generate the following list of process variables.

3.3. Dependency File Example

```
RO LINAC:rec2
LINAC:rec1
PREFIX:rec3
LINAC:rec4 5
RON LINAC:rec5
```

Recall that the third process variable, `PREFIX:rec3`, found in *req3* did not get the benefit of the **#define** found in *req1* and therefore was not transformed into a process variable name found on the IOCs. Assume that these request files were processed by **burtrb** and the generated non-SDDS snapshot file (omitting the header) looks as follows.

```
RO LINAC:rec2 1 222
LINAC:rec1 1 111
LINAC:rec4 5 3 444 555 666
RON LINAC:rec5 1 555
```

Note that `PREFIX:rec3` from *req3* does not appear in the snapshot file. It was not found on the IOCs and was reported as such in the log file (log files explained in section 4). Note also that, although five values were requested for `LINAC:rec4` in *req3*, only three were read.

Consider the following dependency file, *dep*.

dep

```
% contradiction
CONT 1 == 2

% both should fail because not_there
% is not in the database nor is it in the snapshot
CONT ca_get(not_there) < 5
CONT ss_get(not_there) < 5

% should fail because asking for a snapshot
% of a process variable that is NOT a single value
CONT ss_get(LINAC:rec4) != 6

(ss_get(LINAC:rec1)-ca_get(LINAC:rec1)) < 1000
```

We will assume that over time, all the values of all our process variables have changed to 0 on the IOCs. An execution of **burtwb** with the snapshot file and dependency file above, in conjunction with the values on the IOCs as described, would proceed as follows.

The first thing **burtwb** does is read in all the snapshot files. In this case there is only one. The dependency file is then processed by the C preprocessor. Because *dep* does not have **#define** or **#include** directives, this will have no effect. **Burtwb** processes the conditions by automatically writing and compiling a C program based on the conditions and the failure directives. **Burtwb** then executes that program.

In our example, the first condition fails because it is a contradiction. The user is notified that the condition failed in the log file and processing continues as a result of the failure directive CONT. If HALT had been specified instead, the user would have been notified that the condition failed, processing would have terminated here, no other conditions would have been evaluated and nothing would have been written to the IOCs. If no failure directive had been specified, the user would have been notified that the condition failed and prompted whether or not to continue.

The second condition is not even evaluated because the process variable `not_there` is not on any of the IOCs. The user is notified of this in the log file and, again, because the failure directive is CONT, processing continues.

The third condition is also not evaluated because the process variable `not_there` is not in the snapshot file. Again, the user is notified and processing continues.

The fourth condition is not evaluated because, although the process variable `LINAC:rec4` is in the snapshot file, it does not represent a single atomic value. As we mentioned earlier, **ss_get()** only works when the value in the snapshot file is a single atomic value. The user is notified and processing continues.

The last condition is the only one that is evaluated, because all the values can be obtained. The condition (111-0)<1000 is true, so processing simply continues.

The dependency file has been successfully processed, meaning no condition stopped execution. **burtwb** continues by restoring the values in the snapshot file.

4. Log Files

There is a BURT component that did not appear in the overview chapter, log files. Each of the programs that appear in BURT's suite of programs produces a log file each time they are executed. The log file is an ASCII file where BURT places diagnostic information. By default, the information is written to UNIX's *stderr*. This almost always means that by default the log file information will appear on the screen.

The user can control where BURT puts the log file and how much information BURT places into the log file. All of the programs allow users to instruct BURT to place the log file information into a file rather than onto the screen. They also allow the user to increase the amount of information so that BURT tells the *everything* it is doing. By default, BURT gives a general overview about the execution of the program and it prints it on the screen. A full description of how to specify where and how much information appears in the log file is in Chapter 4.

Chapter 4: Using BURT From a UNIX Prompt

BURT can be executed in two ways; from a UNIX prompt passing it arguments on the command line or from its Graphic User Interface (GUI). In this chapter we describe how to run BURT from the UNIX prompt, or command line.

We make two assumptions here. First, that the reader is already familiar with all of BURT's terminology explained above. Second, that the reader is familiar with executing commands at the UNIX prompt that accept arguments on the command line in the form of switches.

1. burtrb

This is BURT's back up tool. It takes as input one or more request files and produces a single Absolute snapshot file.

The user must always specify at least one request file. Request files are specified using the **-f** (for file) option. As with all of BURT's programs, the user may mix SDDS and non-SDDS files freely in a single execution. To specify the two request files *req1* and *req2*, the user would enter the command:

> burtrb -f req1 req2

The snapshot file is written to the UNIX file *stdout*. In most cases this means that the snapshot file is written to the screen. The user can instruct BURT to write the snapshot file to a file, rather than the screen, by using **-o** (for output) option. Extending the example above to writing the snapshot to a file called *snap*, the user would enter the command:

> burtrb -f req1 req2 -o snap

Options can appear in any order. In other words, the above command is equivalent to the following command:

> burtrb -o snap -f req1 req2

This ability to specify options in any order on the command line is true for all of BURT's programs.

The format of the snapshot files can be either SDDS compliant or not. By default, **burtrb** will set the snapshot file format based on the format of the request files. Therefore, if all the request files have the same format, **burtrb** will generate a snapshot file of the same format, otherwise it will generate an SDDS snapshot file. The user always has the option of explicitly specifying the format of the generated snapshot file by specifying either **-sdds** or **-nosdds**.

To explicitly instruct **burtrb** to generate an SDDS snapshot file in our example the user would use the following command:

> burtrb -f req1 req2 -o out -sdds

Whenever creating a snapshot file, the user may augment the snapshot file with comments and/or keywords. These comments and keywords appear in the header portion of the snapshot file. This is done with the **-c** (for comments) and **-k** (for keywords) options. To add the comment ''took this on a good day'' with the keywords ''linac best tune'' the user would use the following command:

> burtrb -f req1 req2 -o out -sdds -c took this on a good day -k linac best tune

The command line can get very long. Occasionally, it is so long that it cannot fit on one line. This is not a problem for BURT. The user must always wait until the entire command line has been typed before hitting the carriage return. Hitting the carriage return sends the command to BURT and hitting the carriage return prematurely will only send the information typed in so far.

By default, **burtrb** writes the terse version of the log file to UNIX's *stderr*, which is typically the screen. The user can instruct **burtrb** to increase the information in the log file by reporting *everything* it is doing by specifying the **-v** (for verbose) option. Additionally, the user can also instruct **burtrb** to place the log file into a file rather than the screen with the **-l** (for log file) option. Here is an example of requesting a verbose log file sent to the file *log* (omitting the comments and keywords above).

> burtrb -f req1 req2 -o out -sdds -v -l log

All the programs in BURT communicate with the IOCs using a facility called channel access [3]. In doing so, it asks channel access to find the values of the process variables on the IOCs. Occasionally, the medium over which channel access communicates becomes saturated with heavy use. This slows down response times and channel access is sometimes fooled into thinking that it cannot find a value because it has taken too long. This rarely happens, but if it does, the user can instruct channel access to try a few more times before giving up. This is done with the **-r** (for retry) option. Specifying retry levels does not slow **burtrb** down, it stops as soon as it finds the value. By default, **burtrb** instructs channel access to try once. The **-r** option instructs it how many *more* times to try. Here is an example of calling **burtrb** asking channel access to try two more times:

> burtrb -f req1 req2 -o out -sdds -v -l log -r 2

Finally, as a diagnostic aid there is a debugging option. All non-SDDS request files are processed by the C pre-processor to interpret any **#define** or **#include** directives that might appear in those files. This is done before **burtrb** gets the files. Each non-SDDS request file is processed by the C pre-processor which generates a new temporary file. This new temporary file reflects the effects of all the **#define** and **#include** statements and it is actually these files that are used as input to **burtrb**.

By default, **burtrb** removes these files at the end of its execution. However, the user can instruct **burtrb** not to remove these files, presumably for later inspection, by specifying the **-d** (for debug) option. When doing so, the user assumes the responsibility of removing the files. Extending our example, the command would be:

> burtrb -f req1 req2 -o out -sdds -v -l log -r 2 -d

It is up to the EPICS system administrator that installs BURT to decide where these files are generated. It is typically the */tmp* directory. The files are named by the loginid of the user calling **burtrb** and the time at which **burtrb** was executed.

loginid_YYMMDD_HHmmSS_I

where

loginid - loginid of user executing BURT
YY - year
MM - month
DD - day
HH - hour
mm - minute
SS - second
I - integer index [0-9]

If they are not in */tmp*, contact the EPICS system administrator and have he/she look at the **#define** macro `PUBLICDIRECTORY` in *burtcommon.h*.

As with all of BURT's programs, simply typing in the name of the program without any arguments places a quick reference usage message onto the screen.

> burtrb

usage: burtrb -f req1 {req2 ...} {-l logfile} {-o outfile} {-d} {-v} {-c ... comments ...} {-k keyword1 ... keywordn} {-r retry_count} {-sdds or -nosdds}

where

-f req1 {req2 ...} - Request file names. This is the only switch that is not optional. You must specify at least one request file.

-l logfile - Log file name. The name of the file where all logging messages (e.g. error messages, reports of process variables that were not found) go. The default is *stderr*.

-o outfile - Snapshot file name. The name of the file where the snapshot information goes. The default is *stdout*.

-d - Debug. Save the files created by processing the request files with the C preprocessor. The default is to delete these files.

-v - Verbose. This increases the amount of information displayed in the logfile.

-c ...comments ... - Comments. Adds comments to the header of the snapshot file.

-k keyword1 ... keywordn - Keywords. Adds keywords to the header of the snapshot file.

-r retry_count - Number of additional attempts to wait for connections. The program will attempt to find all the process variables. If it is unsuccessful, it will try this many more times to establish connections. The default value is 0.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to adopt the SDDS type from the input(s). If there is a heterogenous set of inputs (some SDDS and some non-SDDS), the default is to produce and SDDS compliant snapshot file.

2. burtwb

This is BURT's restore tool. It takes as input one or more snapshot files and zero or more dependency files. Depending on the contents of the snapshot files, it will produce a single Nowrite snapshot file.

The user must always specify at least one snapshot file. Snapshot files are specified using the **-f** (for file) option. As with all of BURT's tools, the user may mix SDDS and non-SDDS files freely in a single execution. To specify the two snapshot files *snap1* and *snap2*, the user would enter the command:

> burtwb -f snap1 snap2

Burtwb takes the contents of the snapshot files and changes the values on the IOCs accordingly. Some of the values in the snapshot files are tagged as RO (Read Only) or RON (Read Only Notify). These values are not restored to the IOCs.

Snapshot files always have a type: Absolute, Relative, or Nowrite. The type of snapshot file instructs **burtwb** how to restore the values in the file to the IOCs. By default, the values in an Absolute snapshot file *replace* the values on the IOCs and the values in a Relative snapshot file are *added* to the values on the IOCs. The values in Nowrite snapshot files do not affect the values on the IOCs.

The user can instruct **burtwb** to treat Absolute snapshot files as Relative and vice versa. In other words, the user can instruct **burtwb** to treat the values in an Absolute snapshot file as though they appeared in a Relative snapshot file, thus adding the values rather than replacing the values, by specifying the **-add** (for add) option. The user can also instruct **burtwb** to treat the values in a Relative snapshot file as though they appeared in an Absolute snapshot file, thus replacing the values rather than adding them, by specifying the **-replace** (for replace) option.

The **-add** option is applied to all the Absolute snapshot files found in the command line, and likewise the **-replace** option is applied to all the Relative snapshot files found in the command line.

These are independent options, meaning the user can specify neither, both, or one of them. To specify the **-add** option only:

> burtwb -f snap1 snap2 -add

Options can appear in any order. In other words, the above command is equivalent to the following command:

> burtwb -add -f snap1 snap2

To specify the **-replace** option only:

> burtwb -f snap1 snap2 -replace

To specify both:

> burtwb -f snap1 snap2 -add -replace

Burtwb produces a single Nowrite snapshot file when it encounters RON values in its set of input snapshot files, otherwise it produces no output file.

The snapshot file is written to the UNIX file *stdout*. In most cases this means that the snapshot file is written to the screen. The user can instruct **burtwb** to write the snapshot file to the a file, rather than the screen, by using **-o** (for output) option. For example, extending the example above, and omitting the **-add** and **-replace** options, writing the snapshot to a file called *nowrite.snap*, the user would enter the command:

> burtwb -f snap1 snap2 -o nowrite.snap

If there are no RON values in *snap1* or *snap2*, then *nowrite.snap* will not be created.

The format of the snapshot files can be either SDDS compliant or not. By default, **burtwb** will set the snapshot file format based on the format of the request files. By default, if all the request files have the same format, **burtwb** will generate a snapshot file of the same format, otherwise it will generate an SDDS snapshot file. The user always has the option of explicitly specifying the format of the generated snapshot file by specifying either **-sdds** or **-nosdds**.

To explicitly instruct **burtwb** to generate an SDDS snapshot file in our example the user would use the following command:

> burtwb -f snap1 snap2 -o nowrite.snap -sdds

Whenever creating a snapshot file, the user has the option of augmenting the snapshot file with comments and/or keywords. These comments and keywords appear in the header portion of the snapshot file. Using the **-c** (for comments) and **-k** (for keywords) options. Adding the comments ‘’restored again’’ with the keywords ‘’restoring linac best tune’’ the user would use the following command:

> burtwb -f snap1 snap2 -o nowrite.snap -sdds -c restored again -k restoring linac best tune

Once again, we see that the command line can get very long. Sometimes it is so long that it cannot fit on one line. This is not a problem for BURT. It is important to remember to wait until you have typed in the entire command line before hitting the carriage return. Hitting the carriage return sends the command to BURT and hitting the carriage return prematurely will only send the information you have typed in so far.

By default, **burtwb** writes the terse version of the log file to UNIX’s *stderr*, which is typically the screen. The user can instruct **burtwb** to increase the information in the log file by reporting *everything* it is doing by specifying the **-v** (for verbose) option. The user can also instruct **burtwb** to place the log file into a file rather than the screen with the **-l** (for log file) option. Here is an example of requesting a verbose log file sent to the file *log* (omitting the comments and keywords above).

> burtwb -f snap1 snap2 -o nowrite.snap -sdds -v -l log

Recall that all the programs in BURT communicate with the IOCs using a facility called channel access. In doing so, it asks channel access to find the values of the process variables from the IOCs. Occasionally, the medium over which channel access communicates becomes saturated with heavy use. This slows down response times and channel access is sometimes fooled into thinking that it cannot find a value because it has taken too long. This rarely happens, but if it does, the user can instruct channel access to try a few more times before giving up. This is done with the -**r** (for retry) option. Specifying retry levels does not slow **burtwb** down, it stops as soon as it finds the value. By default, **burtwb** instructs channel access to try once. The -**r** option tells it how many *more* times to try. Here is an example of calling **burtwb** asking channel access to try two more times:

> burtwb -f snap1 snap2 -o nowrite.snap -sdds -v -l log -r 2

In addition to the snapshot files, the user can specify a set of zero or more dependency files as input to **burtwb** with the **-p** (for predicates) option. These files specify conditions that are tested before **burtwb** restores the snapshots to the IOCs. Extending our example to include dependency files *dep1*, *dep2*, and *dep3*, the call to **burtwb** would look like this:

> burtwb -f snap1 snap2 -o nowrite.snap -sdds -v -l log -r 2 -p dep1 dep2 dep3

Finally, as a diagnostic aid there is a debugging option. **Burtwb** processes dependency files by writing, compiling, and executing a C program on the fly from the conditions specified in the dependency files. Before writing the C program, each dependency file is processed through the C pre-processor to interpret any **#define** or **#include** directives that might appear. Each dependency file is processed this way and each one generates a new temporary file that reflects the effects of all the **#define** and **#include** statements. It is actually these files that **burtwb** uses to write its C program.

By default, **burtwb** removes these temporary files as well as the C source code and the executable program at the end of its execution. However, the user can instruct **burtwb** not to remove these files, presumably for later inspection, by specifying the **-d** (for debug) option. When doing so, the user assumes the responsibility of removing the files. Extending our example to include the debug option, the call to **burtwb** would look like this:

> burtwb -f snap1 snap2 -o nowrite.snap -sdds -v -l log -r 2 -p dep1 dep2 dep3 -d

It is up to the EPICS system administrator that installs BURT to decide where these files are generated. It is typically the */tmp* directory. The files are named by the loginid of the user calling **burtwb** and the time at which **burtwb** was executed.

loginid_YYMMDD_HHmmSS_I

where

loginid - loginid of user who executed BURT
YY - year
MM - month
DD - day
HH - hour
mm - minute
SS - second
I - integer index [0-9]

If they are not in */tmp*, contact the EPICS system administrator and have he/she look at the **#define** macro `PUBLICDIRECTORY` in *burtcommon.h*.

As with all of BURT's programs, simply typing in the name of the program without any arguments places a quick reference usage message onto the screen.

> burtwb

usage: burtwb -f snap1 {snap2 ...} {-l logfile} {-o outfile} {-c ... comments ...}
{-k keyword1 ... keywordn} {-d} {-v} {-p dep1 ... depn} {-r retry count} {-add} {-replace} {-sdds or -nosdds}

where

-f snap1 {snap2 ...} - Snapshot file names. This is the only switch that is not optional. You must specify at least one snapshot file.

-l logfile - Log file name. The name of the file where all logging messages (e.g. error messages, reports of process variables that were not found) go. The default is *stderr.*

-o outfile - Snapshot file name. If any of the snapshot files read only notify values, this file is created and those values are placed there. If none of the snapshot files have read only notify values, then no file is created. The default is *stdout*.

-c ... comments ... - Comments. Adds comments to the header of the snapshot file.

-k keyword1 ... keywordn - Keywords. Adds keywords to the header of the snapshot file.

-d - Debug. Save the files created by processing the dependency files with the C preprocessor. The default is to delete these files.

-v - Verbose. This increases the amount of information displayed in the logfile.

-p dep1 ... depn - Dependency file names. The names of the dependency files containing predicates to be evaluated before writing the values from the snapshot files.

-r retry count - Number of additional attempts to wait for connections. The program will attempt to find all the process variables. If it is unsuccessful, it will try this many more times to establish connections. The default value is 0.

-add - Absolute snapshots written as adds. All the absolute snapshots, i.e., those taken directly off IOCs, will be written as additions to the values found on the IOCs. The default is to write the absolute snapshots as replacement values on the IOCs.

-replace - Relative snapshots written as replacements. All the relative snapshots, i.e., those generated by adding or subtracting two snapshots, will be written to replace the values on the IOCs. The default is to write the relative snapshots as additions to the values on the IOCs.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to adopt the SDDS type from the input(s). If there is a heterogenous set of inputs (some SDDS and some non-SDDS), the default is to produce and SDDS compliant snapshot file.

3. burtmath

This is BURT's arithmetic tool. It allows the user to add or subtract the contents of two snapshot files producing a Relative snapshot file. It also can multiply the contents of a single snapshot file by a scalar constant producing a snapshot file of the same type. In this case, the output snapshot file inherits its type (Absolute, Relative, or Nowrite) from the input snapshot file.

When adding or subtracting snapshot files, only like terms will be added and subtracted. This means that only those process variables with the same name and same number of elements that appear in both input snapshot files will appear in the output snapshot file. For devices, only those that have the same device name and same lineage that appear in both input snapshot files will appear in the output snapshot file.

When supplying two snapshot files, the default operation is subtraction. Determining the difference of two snapshot files *snap1*-*snap2* can be done:

> burtmath snap1 snap2

or

> burtmath snap1 snap2 -sub

Determining the sum of two snapshot files *snap1+snap2* can be done:

> burtmath snap1 snap2 -add

When supplying one snapshot file, the user must use the **-mult** (for multiplication) option. By default **burtmath** will multiply the values in the snapshot file by the scalar constant 1. Here is an example:

> burtmath snap1 -mult

which is equivalent to

> burtmath snap1 -mult 1

which, in turn, is equivalent to

> burtmath snap1 -mult 1.0

Optionally, the user can the **-mult** option and supply a scalar constant other than 1. Here is an example multiplying the values in *snap1* by the scalar constant 2.4:

> burtmath snap1 -mult 2.4

The snapshot file is written to the UNIX file *stdout*. In most cases this means that the snapshot file is written to the screen. The user can instruct BURT to write the snapshot file to the a file, rather than the screen, by using **-o** (for output) option. For example, extending the example above to write the snapshot to a file called *snap.sum*, the user would enter the command:

> burtmath snap1 snap2 -add -o snap.sum

Options can appear in any order. In other words, the above command is equivalent to the following command:

> burtmath snap1 snap2 -o snap.sum -add

Note when using **burtmath** the filenames must always appear after the word **burtmath**. The filenames are not options here. Everything else on the command line is an option and may appear in any order. This ability to specify options in any order on the command line is true for all of BURT's programs.

The format of the snapshot files can be either SDDS compliant or not. By default, **burtmath** will set the snapshot file format based on the format of the input snapshot file(s). By default, if all the snapshot files have the same format, **burtmath** will generate a snapshot file of the same format, otherwise it will generate an SDDS snapshot file. The user always has the option of explicitly specifying the format of the generated snapshot file by specifying either **-sdds** or **-nosdds**.

To explicitly instruct **burtmath** to generate an SDDS snapshot file in our example we would use the following command:

> burtmath snap1 snap2 -add -o snap.sum -sdds

Whenever creating a snapshot file, the user has the option of augmenting the snapshot file with comments and/or keywords. These comments and keywords appear in the header portion of the snapshot file. This is done by using the **-c** (for comments) and the **-k** (for keywords) options. Adding the comment ‘’trying something goofy’’ with the keywords ‘’linac sum’’ the user would use the following command:

> burtmath snap1 snap2 -add -o snap.sum -sdds -c trying something goofy -k linac sum

Once again, we see that the command line can get very long. Sometimes it is so long that it cannot fit on one line. This is not a problem for BURT. It is important to remember to wait until you have typed in the entire command line before hitting the carriage return. Hitting the carriage return sends the command to BURT and hitting the carriage return prematurely will only send the information you have typed in so far.

By default, **burtmath** writes the terse version of the log file to UNIX's *stderr*, which is typically the screen. The user can instruct **burtmath** to increase the information in the log file by reporting *everything* it is doing by specifying the **-v** (for verbose) option. The user can also instruct **burtmath** to place the log file into a file rather than the screen with the **-l** (for log file) option. Here is an example of requesting a verbose log file sent to the file *log* (omitting the comments and keywords above).

> burtmath snap1 snap2 -add -o snap.sum -sdds -v -l log

As with all of BURT's programs, simply typing in the name of the program without any arguments places a quick reference usage message onto the screen.

> burtmath

usage: **burtmath (snap1 snap2 {-add or -sub}) or (snap1 -mult m) {-l logfile}**
{-o outfile} {-v} {-c ... comments ...} {-k keyword1 ... keywordn}
{-sdds or -nosdds}

where

snap1 snap2 {-add or -sub} - Adding/subtracting snapshots. Here you wish to either add or subtract exactly two snapshot files. You must specify exactly two snapshot files when performing either addition or subtraction. The default operation is -sub (subtraction).

snap1 -mult m - Multiplying snapshots. Here you wish to multiply all the values in snap1 by multiplication factor m.

-l logfile - Log file name. The name of the file where all logging messages (e.g. error messages, reports of process variables that were not found) go. The default is *stderr*.

-o outfile - Result file name. The name of the file where the resulting information goes. The default is *stdout*.

-v - Verbose. This increases the amount of information displayed in the logfile.

-c ... comments ... - Comments. Adds comments to the header of the result file.

-**k keyword1 ... keywordn** - Keywords. Adds keywords to the header of the result file.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to adopt the SDDS type from the input(s). If there is a heterogenous set of inputs (some SDDS and some non-SDDS), the default is to produce and SDDS compliant snapshot file.

4. burtset

This is BURT's set operation program. It performs set operations (union, intersection, and difference) on a pair of request files and outputs a request file.

The user must specify exactly two request files. This is done with the **-f** (for file) option. As with all of BURT's programs, the user may mix SDDS and non-SDDS files freely in a single execution. To specify the two request files *req1* and *req2*, the user would enter the command:

> burtset -f req1 req2

The default set operation is to perform a union of the two files. That is, produce a request file that has those process variables and devices that appear in either input request file. The above command is therefore equivalent to the command:

> burtset -f req1 req2 -union

Options can appear in any order. In other words, the above command is equivalent to the following command:

> burtset -union -f req1 req2

The result of a union operation is that all the names that appear in both of the input files appear in the output file. If a process variable or device appears in exactly one file, it will appear in the output file. If a process variable or device appears in both files, it will appear once in the output file. In this case it will be tagged RON if it is RON in either source file. If it is not RON in either file, it will be tagged RO if it is RO in either source file. Otherwise it will not be tagged at all. If the process variable or device appears in both files with a different number of elements in both, **burtset** arbitrarily chooses one of the number of element requests and propagates that to the output file.

The user may request an intersection be taken. That is, produce a request file that has those process variables and devices found in both input request files. To request an intersection of the request files *req1* and *req2*:

> burtset -f req1 req2 -inter

This intersection match is done on name only, it ignores any tagging or number of requested elements. It propagates tagging based on maximizing severity. In other words, if there is a match then the tag of the output will be RON if either source is RON. If neither is RON, then the tag of the output will be RO if either is RO. Otherwise, the output will not be tagged. Additionally, in the case of the match, **burtset** arbitrarily chooses the number of requested for the output.

The user may request a difference be taken. That is, produce a request file that has those process variables and devices found in *req1* but not *req2*. To request a difference *req1-req2*:

> burtset -f req1 req2 -diff

Matching is done with respect to name only, i.e.,ignoring tagging or number of requested. As the operation implies, the output file will be a subset of the first request file.

Independent of the chosen set operation, the resulting request file is written to the UNIX file *stdout*. In most cases this means that the request file is written to the screen. The user can instruct BURT to write the request file to the a file, rather than the screen, by using **-o** (for output) option. Extending the example above to write the request to a file called *union.req*, the user would enter the command:

> burtset -f req1 req2 -union -o union.req

The format of the output request file can be either SDDS compliant or not. By default, **burtset** will set the output request file format based on the format of the input request files. By default, if all the input request files have the same format, **burtset** will generate a request file of the same format, otherwise it will generate an SDDS request file. The user always has the option of explicitly specifying the format of the generated request file by specifying either **-sdds** or **-nosdds**.

To explicitly instruct **burtset** to generate an SDDS request file in our example the user would use the following command:

> burtset -f req1 req2 -union -o union.req -sdds

By default, **burtset** writes the terse version of the log file to UNIX's *stderr*, which is typically the screen. The user can instruct **burtset** to increase the information in the log file by reporting *everything* it is doing by specifying the **-v** (for verbose) option. Additionally, the user can also instruct **burtrb** to place the log file into a file rather than the screen with the **-l** (for log file) option. Here is an example of requesting a verbose log file sent to the file *log*.

> burtset -f req1 req2 -union -o union.req -sdds -v -l log

Finally, as a diagnostic aid there is a debugging option. All non-SDDS request files are processed by the C pre-processor to interpret any **#define** or **#include** directives that might appear in those files. This is done before **burtset** gets the files. Each non-SDDS request file is processed by the C pre-processor which generates a new temporary file. This new temporary file reflects the effects of all the **#define** and **#include** statements and it is actually these files that are used as input to **burtset**.

By default, **burtset** removes these files at the end of its execution. However, the user can instruct **burtset** not to remove these files, presumably for later inspection, by specifying the **-d** (for debug) option. When doing so, the user assumes the responsibility of removing the files. Extending our example, the command would be:

> burtset -f req1 req2 -union -o union.req -sdds -v -l log -d

It is up to the EPICS system administrator that installs BURT to decide where these files are generated. It is typically the */tmp* directory. The files are named by the loginid of the user calling **burtset** and the time at which **burtset** was executed.

loginid_YYMMDD_HHmmSS_I

where

loginid - loginid of user who executed BURT
YY - year
MM - month
DD - day
HH - hour
mm - minute
SS - second
I - integer index [0-9]

If they are not in */tmp*, contact the EPICS system administrator and have he/she look at the **#define** macro `PUBLICDIRECTORY` in *burtcommon.h*.

As with all of BURT's programs, simply typing in the name of the program without any arguments places a quick reference usage message onto the screen.

> burtset

usage: **burtset -f req1 req2 {-union or -inter or -diff} {-l logfile} {-o outfile} {-d} {-v} {-sdds or -nosdds}**

where

-f req1 req2 - Request filenames. This is the only switch that is not optional. You must specify at least two request files.

-union or -inter or -diff - Set operation. The set operation (union, intersection, or difference) to be performed on the two request files. The default is union.

-l logfile - Log file name. The name of the file where all logging messages (e.g. error messages) go. The default is *stderr*.

-o outfile - Request file name. The name of the file where the result of the set operation goes. The default is *stdout*.

-d - Debug. Save the files created by processing the request files with the C preprocessor. The default is to delete these files.

-v - Verbose. This increases the amount of information displayed in the logfile.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to adopt the SDDS type from the inputs. If there is a heterogenous set of inputs (one SDDS and the other non-SDDS), the default is to produce and SDDS compliant snapshot file.

5. burtconvertsnap

This is BURT's snapshot conversion tool. It takes as input one snapshot file and produces a single snapshot file. Its purpose is to convert SDDS snapshot files to non-SDDS snapshot files and vice versa.

Converting snapshot files in this way can alter the informational contents (not just the syntax) of the files. Recall that SDDS snapshot files may have devices in them while non-SDDS snapshot files cannot. This poses a problem for **burtconvertsnap** when it is asked to convert an SDDS snapshot file containing devices to a non-SDDS snapshot file. **Burtconvertsnap** solves this problem by translating all the devices in the SDDS snapshot file into their process variable constituents. It then places the process variable name along with its associated value, as opposed to the device name and its value, into the output non-SDDS snapshot file. Additionally, the lineage of those devices found in the SDDS snapshot file is lost as well.

The informational difference between input and output snapshot files only occurs when converting SDDS snapshot files containing devices to non-SDDS snapshot files. Converting non-SDDS snapshot files or SDDS snapshot files without devices always produces informationally equivalent files.

The user must always specify exactly one snapshot file as input. To specify the snapshot file *snap.source* the user would enter the following command:

> burtconvertsnap snap.source

By default, **burtconvertsnap** will produce a snapshot file with a different format than the input file. The user has the option of explicitly specifying the type of snapshot file should be produced by using either the -sdds or -nosdds option. Extending the example above to instruct **burtconvertsnap** to produce an SDDS snapshot file, the user would enter the command:

> burtconvertsnap snap.source -sdds

The converted snapshot file is written to the UNIX file *stdout*. In most cases this means that the snapshot file is written to the screen. The user can instruct BURT to write the snapshot file to a file, rather than the screen, by using the **-o** (for output) option. Extending the example above, writing the snapshot to a file called *snap.converted* the user would enter the command:

> burtconvertsnap snap.source -sdds -o snap.converted

Options can appear in any order. In other words, the above command is equivalent to the following command:

> burtconvertsnap snap.source -o snap.converted -sdds

Note that the specification of the input snapshot file is *not* an option. Only those command line arguments that begin with ‘‘-’’ are considered options. For that reason, the name of the input snapshot file must always appear immediately after the command **burtconvertsnap**.

By default, **burtconvertsnap** writes the terse version of the log file to UNIX’s *stderr*, which is typically the screen. The user can instruct **burtconvertsnap** to increase the information in the log file by reporting everything it is doing by specifying the **-v** (for verbose) option. Additionally, the user can also instruct **burtconvertsnap** to place the log file into a file rather than the screen with the **-l** (for log file) option. Here is an example requesting a verbose log file sent to the file *log*.

> burtconvertsnap snap.source -o snap.converted -sdds -v -l log

As with all of BURT’s programs, simply typing in the name of the program without any arguments places a quick reference usage message onto the screen.

> burtconvertsnap

usage: **burtconvertsnap snap {-l logfile} {-o outfile} {-v} {-sdds or -nosdds}**

where

snap - Snapshot filename. This is the only switch that is not optional. You must specify exactly one snapshot file. This is the snapshot file that will be converted.

-l logfile - Log filename. The name of the file where all logging messages (e.g. error messages) go. The default is *stderr.*

-o outfile - Output snapshot filename. This is where the newly generated converted snapshot file will be placed. The default is *stdout*.

-v - Verbose. This increases the amount of information displayed in the logfile.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to produce whatever the input snapshot file is not.

Chapter 5: Using BURT From Its GUI

BURT can be executed in two ways; from a UNIX prompt passing it arguments on the command line or from its Graphic User Interface (GUI). In this chapter we describe how to run BURT from its GUI. We assume that the reader is already familiar with all of BURT's terminology explained above.

1. Main

Below is BURT's top level GUI window. It is created when users enter the command **burtgooey** at a UNIX prompt. It is from this window that users invoke BURT's functions as well as exiting the BURT GUI. Each of the buttons on this window (with the exception of the "Done" button) creates other windows that provide access to BURT's programs, e.g., **burtrb**, **burtwb**, etc.

The "Backup" button creates a window that provides access to **burtrb** while the "Restore" button provides access to **burtwb**. The "Add/Sub" and "Mult" buttons each create their own window yet each provides access to **burtmath**. The "Set" button creates a window that provides access to **burtset**. The "Done" button terminates the GUI session.

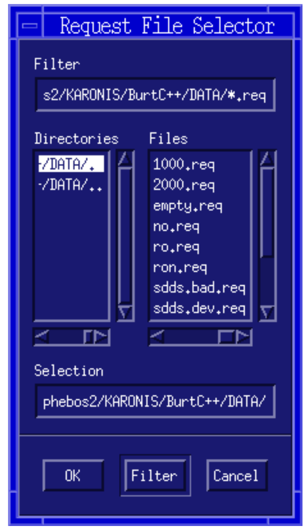
The windows that each button creates are nothing more than a convenient way to construct a UNIX command that calls one of BURT's programs. Operating BURT from its GUI is no more powerful than operating it from a UNIX prompt, and further, understanding the command line options for executing BURT from a UNIX prompt will greatly enhance a user's ability and understanding in executing BURT from its GUI. This will become more obvious as we explore each of the windows in turn.

2. Backup

The window below is created when the "Backup" button is pressed in the top level window. The purpose of this window is to construct a **burtrb** command and pass it on to UNIX for execution.

At the top of this window is a box. It is the list of request files that are to be supplied to **burtrb**. Initially the list is empty. The five buttons immediately below the list, starting with "Request Files ..." and ending with "Print Selected" all pertain to that list. Users are not allowed to type in the box. Information is added/deleted to/from the list by using the buttons beneath it.

Request files are added to the list using the first button, "Request Files ...". This button creates a file selection window, exactly like the one below, from which request files can be selected and added.

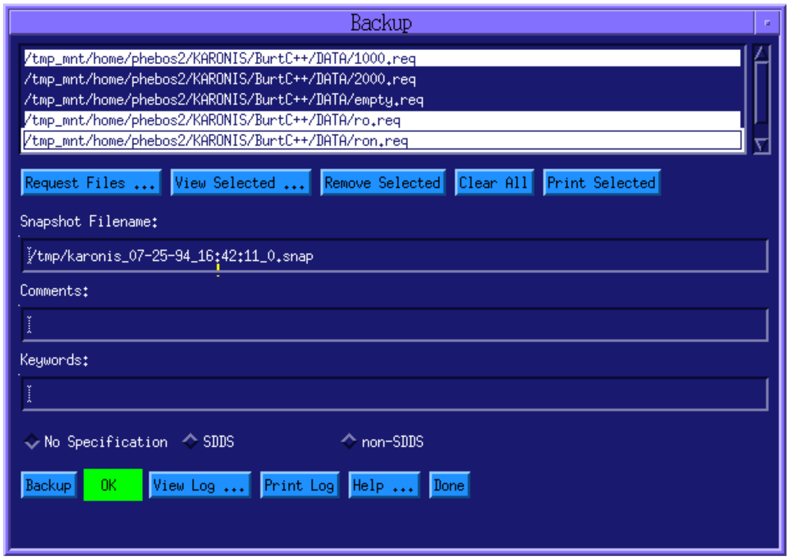


Files that have been selected with this window are added to the list. When a file is added to the list, it is added with its full UNIX pathname. Once a file appears in the list, attempting to add it again will have no effect. In other words, a file may appear at most one time in the list.

Here is an example with a list that has a few members in it.

Note that when the number of files that appear in the list exceeds the size of the box a scrollbar automatically appears to the right.

It is possible to select particular files from the list. This is done by positioning the cursor over the desired file and clicking the left mouse button. When a file is selected it becomes highlighted. Clicking the left mouse button on a different file deselects the original file and selects the new file. To select more than one file, click the left mouse over the first file to select it. To select the additional files, click the left mouse button over them while holding down the control key on the keyboard. Here is an example with three files selected.



File selection pertains to the buttons "View Selected ...", "Remove Selected", and "Print Selected" only. They do not effect which request files will be used in the backup. All files that appear in the list will be used in the backup, whether they are selected or not.

Adding a file to this list does not insure that it is a request file. BURT does not check that each file brought into the list is a valid request file. Users are therefore provided a facility to inspect any of the files that appear in the list with the "View Selected ..." button. Pressing this button will create a new window displaying the selected files. Only ASCII files, i.e., files containing only printable characters, will be displayed. The "Print Selected" acts just like the "View Selected ..." button except that the files are sent to the printer instead of appearing in a newly created window. Files may be removed from the list using either the "Remove Selected" or the "Clear All" buttons.

Recall that BURT's backup program generates a snapshot file. The user must always supply a snapshot filename. When the backup window is created, a default snapshot filename is supplied in the text field labeled "Snapshot Filename:". The user may change this to a non-NULL filename. The next two text fields, labeled "Comments:" and "Keywords:", are where the user may specify any optional comments and/or keywords he would like to augment the snapshot file with.

The user has the option of specifying the format of the generated snapshot file with the panel of radio buttons labeled "No Specification", "SDDS", and "non-SDDS". Selecting "No Specification" will generate a snapshot file that is of the same type as the request file(s). If there is more than one request file and they are of different types, an SDDS snapshot file will be generated.

The top portion of the window describes how to *configure* a backup. The bottom row of buttons (with the exception of the "Done" button) pertains to *executing* the backup. Once the appropriate files have placed in the request file list, a snapshot filename has been provided, any comments and/or keywords have been provided, and a snapshot file format has been selected the user is ready to perform the backup. This is done with the "Backup" button.

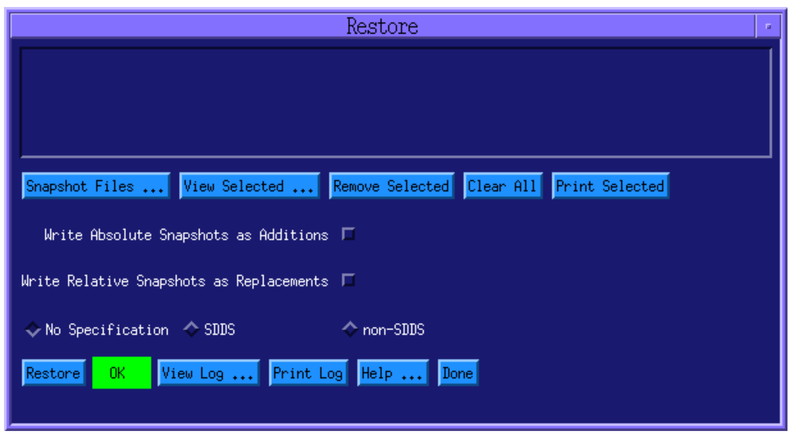
Next to the "Backup" button is a status indicator, it is not a button. Initially it is green and displays the word "OK". Pressing the "Backup" button initiates the backup process. During that process the status indicator turns blue. At the end of the process the status indicator returns to green and displays the word "OK" if everything went all right. If something went wrong, the status indicator turns red and displays the words "NOT OK".

Recall that all BURT operations generate a log. To view this log, after executing a backup and receiving a "NOT OK" status for example, the user may press the "View Log ..." button. This will generate a new window containing the verbose version of BURT's backup log file. The log may be printed using the "Print Log" button. A new log is generated each time the "Backup" button is pressed, thus destroying the previous log.

At the end of the backup operations, the user may press the "Done" button which removes the backup window from the screen.

3. Restore

The window below is created when the "Restore" button is pressed in the top level window. The purpose of this window is to construct a **burtwb** command and pass it on to UNIX for execution.

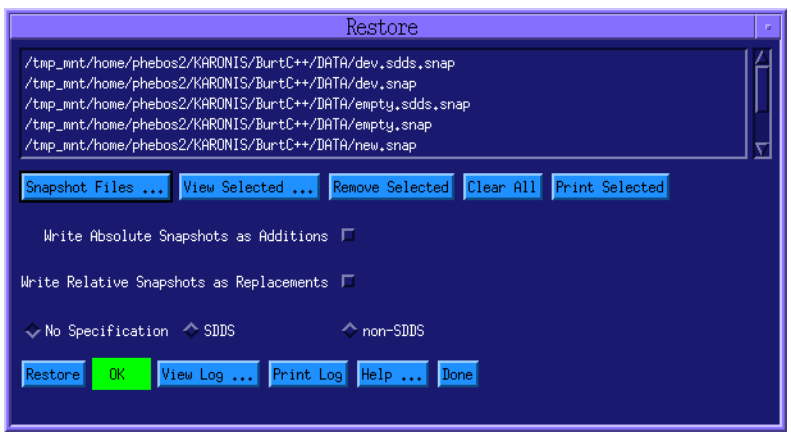


At the top of this window is a box. It is the list of snapshot files that are to be supplied to **burtwb**. Initially the list is empty. The five buttons immediately below the list, starting with "Snapshot Files ..." and ending with "Print Selected" all pertain to that list. Users are not allowed to type in the list. Information is added/deleted to/from the list using the buttons beneath it.

Snapshot files are added to the list using the first button, "Snapshot Files ...". This button creates a file selection window, exactly like the one below, from which snapshot files can be selected and added.

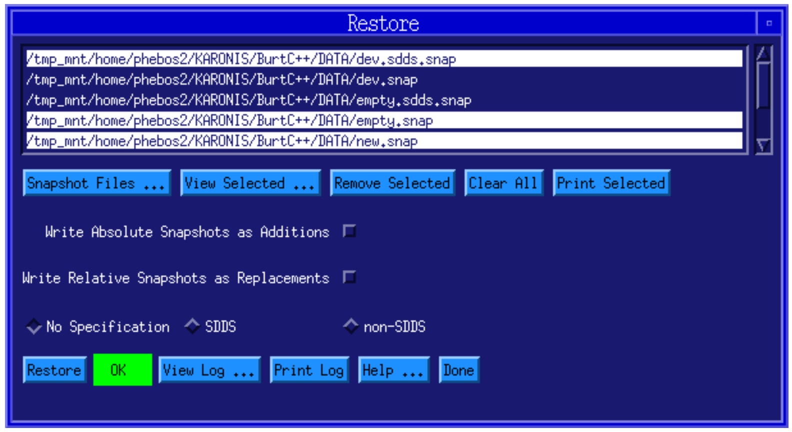
Files that have been selected with this window are added to the list. When a file is added to the list, it is added with its full UNIX pathname. Once a file appears in the list, attempting to add it again will have no effect. In other words, a file may appear at most one time in the list.

Here is an example with a list that has a few members in it.



Note that when the number of files that appear in the list exceeds the size of the box a scrollbar automatically appears to the right.

It is possible to select particular files from the list. This is done by positioning the cursor over the desired file and clicking the left mouse button. When a file is selected it becomes highlighted. Clicking the left mouse button on a different file deselects the original file and selects the new file. To select more than one file, click the left mouse over the first file to select it. To select the additional files, click the left mouse button over them while holding down the control key on the keyboard. Here is an example with three files selected.



File selection pertains to the buttons "View Selected ...", "Remove Selected", and "Print Selected" only. It does not effect which snapshot files will be used in the restoration. All the files that appear in the list will be used in the restoration, whether they are selected or not.

Adding a file to this list does not insure that it is a snapshot file. BURT does not check that each file brought into the list is a valid snapshot file. Users are therefore provided a facility to inspect any of the files that appear in the list with the "View Selected ..." button. Pressing this button will generate a new window displaying the selected files. Only ASCII files, i.e., files containing only printable characters, will be displayed. The "Print Selected" button acts just like the "View Selected ..." button except that the files are sent to the printer instead of appearing in a newly created window. Files may be removed from the list using either the "Remove Selected" or the "Clear All" buttons.

The next two switches instruct BURT how to restore the different types of snapshot files it encounters in the list. Enabling the first switch, labeled "Write Absolute Snapshots as Additions", instructs BURT to restore Absolute snapshots as though they were Relative snapshots. Enabling the second switch, labeled "Write Relative Snapshots as Replacements", instructs BURT to restore Relative snapshots as though they were Absolute (see Chapter 3 Section 2.1).

Recall that BURT's restoration program occasionally generates a Nowrite snapshot file. This happens whenever there are RON values on any of the input snapshot files. The user has the option of specifying the format of the generated snapshot file with the panel of radio buttons labeled "No Specification", "SDDS", and "non-SDDS". Selecting "No Specification" will generate a snapshot file that is of the same type as the input snapshot file(s). If there is more than one input snapshot file and they are of different types, an SDDS snapshot file will be generated.

The generated snapshot is not placed into a file. Rather, it appears in a window as a result of starting the restoration process. If there are no RON values in the input snapshot files, then no such window is generated. Below is an example of a window that is generated as a result of executing a restoration that did contain RON values. The format of the generated snapshot is non-SDDS.

This window is dismissed with the “Done” button. The contents of the window may be printed with the “Print” button.

The preceding discussion of the top portion of the restoration window describes how to *configure* a restoration. The bottom row of buttons (with the exception of the “Done” button) pertains to *executing* the restoration. Once the appropriate files been placed in the snapshot file list, how to write the snapshot files has been determined, and the format of the potentially generated snapshot has been set the user is ready to perform the restoration. This is done with the “Restore” button.

Next to the “Restore” button is a status indicator, it is not a button. Initially it is green and displays the word “OK”. Pressing the “Restore” button initiates the restoration process. During that process the status indicator turns blue. At the end of the process the status indicator returns to green and displays the word “OK” if everything went all right. If something went wrong, the status indicator turns red and displays the words “NOT OK”.

Recall that all BURT operations generate a log. To view this log, after executing a restoration and receiving a “NOT OK” status for example, the users may press the “View Log ...” button. This will generate a new window containing the verbose version of BURT’s restoration log. The log may be printed using the “Print Log” button. A new log is generated each time the “Restore” button is pressed, thus destroying the previous log.

At the end of the restore operations, the user may press the “Done” button which removes the restore window from the screen.

4. Add/Sub

The window below is created when the “Add/Sub” button is pressed in the top level window. The purpose of this window is to construct a **burtmath** command for the addition or subtraction of snapshot files and pass it on to UNIX for execution.

Both the addition and subtraction operations take exactly two snapshot files as input. The name of these two snapshot files must appear in the top two text fields. Users may select snapshot files by using the button labeled "Snapshot Files ...". Two such buttons have been provided, one for each snapshot file. These buttons create a file selection window from which users may select a snapshot file. The file selection window for the first request file appears below.

Selecting a snapshot file places the selected file into the appropriate text field. Below is an example of the Addition/Subtraction window with both snapshot files specified.

Placing filenames into these text fields does not insure that they are valid snapshot files. BURT does not check that each is a valid snapshot file. Users are therefore provided a facility to inspect the files with the “View ...” buttons. Pressing this button generates a window in which the file appears. Only ASCII files, i.e., files containing only printable characters, can be viewed this way. The “Print” button acts just like the “View ...” button except that the files are sent to the printer rather than appearing in a newly created window.

The operation, either addition or subtraction, is selected by the panel of radio buttons, labeled “Addition” and “Subtraction”, that appear between the two snapshot file names.

Recall that BURT’s arithmetic program generates a snapshot file. The user must always supply a snapshot filename. When the Addition/Subtraction window is created, a default snapshot filename is supplied in the text field labeled “Snapshot Filename:”. The next two fields, labeled “Comments:” and “Keywords:”, are where the user may specify any optional comments and/or keywords he would like to augment the snapshot file with.

The user has the option of specifying the format of the generated snapshot file with the panel of radio buttons labeled “No Specification”, “SDDS”, and “non-SDDS”. Selecting “No Specification” will generate a snapshot file that is of the same type as the two input snapshot files if they are of the same type. If they are not the same type, an SDDS snapshot file will be generated.

The preceding discussion of the top portion of the window describes how to *configure* an addition or subtraction. The bottom row of buttons (with the exception of the “Done” button) pertains to *executing* the addition or subtraction. Once the two snapshot files have been selected, the operation has been chosen, an output snapshot filename has been chosen, any comments and/or keywords have been provided, and an output snapshot file format has been selected the user is ready to perform the arithmetic operation. This is done with the “Go” button.

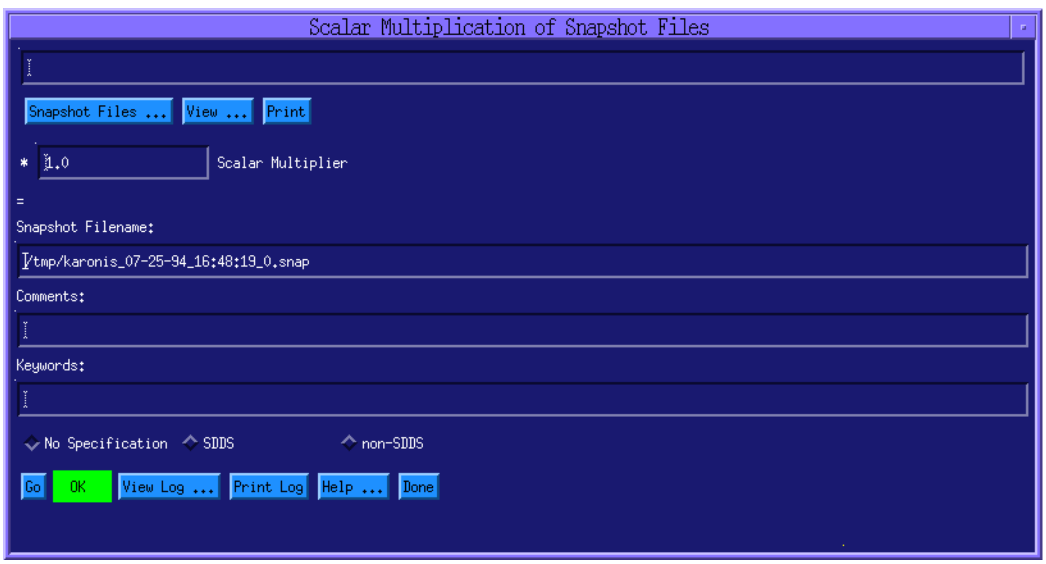
Next to the “Go” button is a status indicator, it is not a button. Initially it is green and displays the word “OK”. Pressing the “Go” button initiates the arithmetic operation. During that operation the status indicator turns blue. At the end of the operation the status indicator returns to green and displays the word “OK” if everything went all right. If something went wrong, the status indicator turns red and displays the words “NOT OK”.

Recall that all BURT operations generate a log. To view this log, after performing an operation and receiving a “NOT OK” status for example, the user may press the “View Log ...” button. This will generate a window containing the verbose version of BURT’s arithmetic operation log. The log may be printed using the “Print Log” button. A new log is generated each time the ‘’Go” button is pressed, thus destroying the previous log.

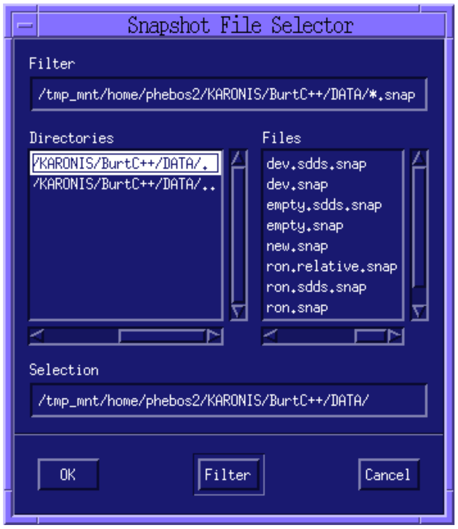
At the end of the addition and/or subtraction operations, the user may press the “Done” button to remove the addition/subtraction window from the screen.

5. Mult

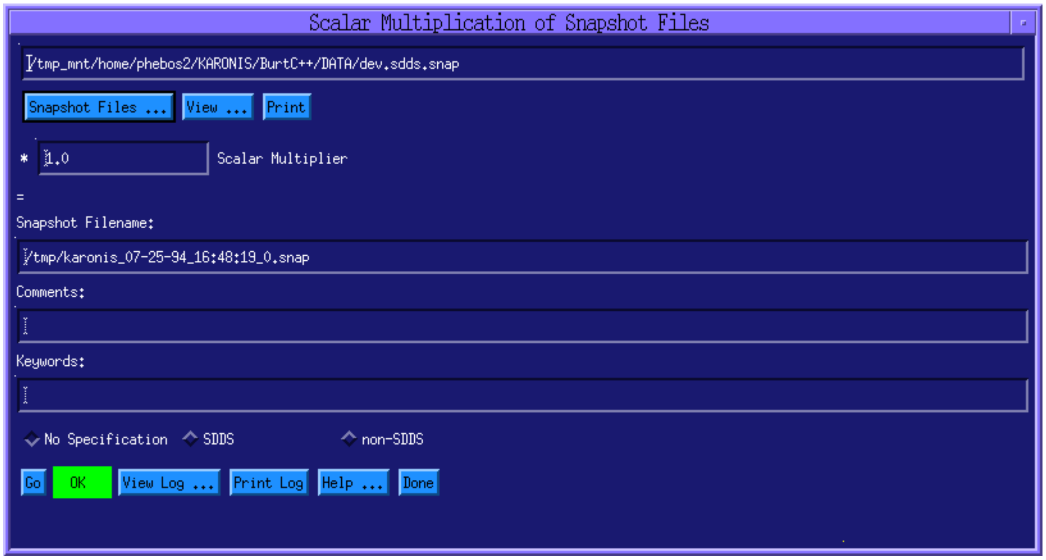
The window below is created when the “Mult” button is pressed in the top level window. The purpose of this window is to construct a **burtmath** command for the multiplication of snapshot files by a scalar constant and passing it on to UNIX for execution.



The multiplication operation takes exactly one snapshot file as input. The name of this snapshot file must appear in the top text field. Users may select snapshot files by using the button labeled “Snapshot Files ...”. This button creates a file selection window from which users may select a snapshot file, just like the one below.



Selecting a snapshot file places the selected file into the text field. Below is an example with a snapshot file specified.



Placing a filename into the text field does not insure that it is a valid snapshot file. BURT does not check that it is a valid snapshot file. Users are therefore provided a facility to inspect the file with the "View ..." button. Pressing this button generates a window in which the file appears. Only ASCII files, i.e., files containing only printable characters, can be viewed this way. The "Print" button acts just like the "View ..." button except that the files are sent to the printer rather than appearing in a newly created window.

The next text field, labeled "Scalar Multiplier" is where the user specifies the scalar constant to multiply the snapshot file by. By default this value is 1.0.

Recall that BURT's arithmetic program generates a snapshot file. The user must always supply a snapshot filename. When the Scalar Multiplication window is created, a default snapshot filename is supplied in the text field labeled "Snapshot Filename:". The user may change this filename. The next two fields, labeled “Comments:” and “Keywords:”, are where the user may specify any optional comments and/or keywords he would like to augment the snapshot file with.

The user has the option of specifying the format of the generated snapshot file with the panel of radio buttons labeled “No Specification”, “SDDS”, and “non-SDDS”. Selecting “No Specification” will generate a snapshot file that is of the same type as the input snapshot file.

The preceding discussion of the top portion of the window describes how to *configure* scalar multiplication. The bottom row of buttons (with the exception of the “Done” button) pertains to *executing* scalar multiplication. Once the input snapshot file has been selected, the scalar constant specified, an output snapshot filename has been chosen, any comments and/or keywords have been provided, and an output snapshot file format has been selected the user is ready to perform the arithmetic operation. This is done with the “Go” button.

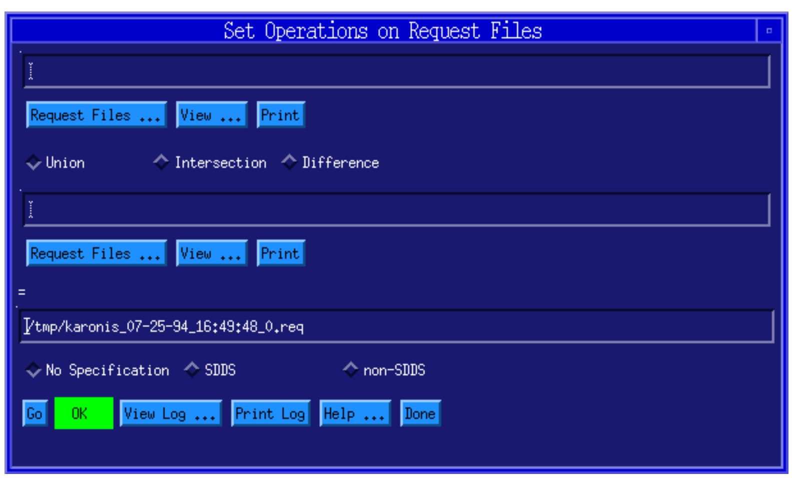
Next to the “Go” button is a status indicator, it is not a button. Initially it is green and displays the word “OK”. Pressing the “Go” button initiates the arithmetic operation. During that operation the status indicator turns blue. At the end of the operation the status indicator returns to green and displays the word “OK” if everything went all right. If something went wrong, the status indicator turns red and displays the words “NOT OK”.

Recall that all BURT operations generate a log. To view this log, after performing an operation and receiving a “NOT OK” status for example, the user presses the “View Log ...” button. This will generate a new window containing the verbose version of BURT’s arithmetic operation log. The log may be printed using the “Print Log” button. A new log is generated each time the ‘’Go” button is pressed, thus destroying the previous log.

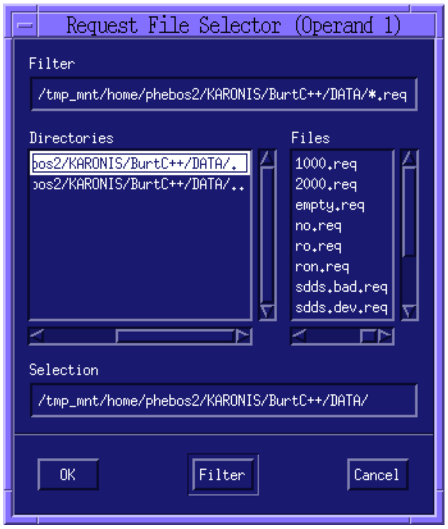
At the end of the scalar multiplication operations, the user may press the “Done” button which removes the scalar multiplication window from the screen.

6. Set

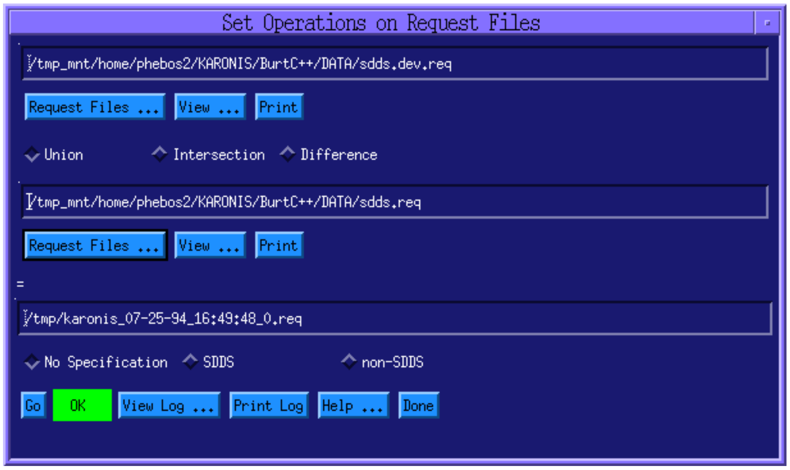
The window below is created when the “Set” button is pressed in the top level window. The purpose of this window is to construct a **burtset** command and pass it on to UNIX for execution.



The set operation takes exactly two request files as input. The name of these two request files must appear in the top two text fields. Users may select request files by using the button labeled “Request Files ...”. Two such buttons have been provided, one for each request file. These buttons create a file selection window from which users may select a request file. The file selection window for the first request file appears below.



Selecting a request file places the selected file into the appropriate text field. Below is an example of the Set window with both request files specified.



Placing filenames into these text fields does not insure that they are valid request files. BURT does not check that each file is a valid request file. Users are therefore provided a facility to inspect the files with the “View ...” buttons. Pressing this button generates a window in which the file appears. Only ASCII files, i.e., files containing only printable characters, can be viewed this way. The “Print” button acts just like the “View ...” button except that the files are sent to the printer rather than appearing in a newly created window.

The set operation is selected by the panel of radio buttons, labeled “Union”, “Intersection”, and “Difference”, that appear between the two request filenames.

Recall that BURT’s set program generates a request file. The user must always supply a request filename. When the Set window is created, a default request filename is supplied in the text field beneath the “=” sign. The user may change this filename.

The user has the option of specifying the format of the generated request file with the panel of radio buttons labeled ‘‘No Specification’’, ‘‘SDDS’’, and ‘‘non-SDDS’’. Selecting ‘‘No Specification’’ will generate a request file that is of the same type as the two input request files if they are of the same type. If they are not the same type, an SDDS request file will be generated.

The preceding discussion of the top portion of the window describes how to *configure* a set operation. The bottom row of buttons (with the exception of the ‘‘Done’’ button) pertains to *executing* the set operation. Once the two request files have been selected, the operation chosen, an output snapshot filename has been chosen, any comments and/or keywords have been provided, and an output request file format has been selected the user is ready to perform the set operation. This is done with the ‘‘Go’’ button.

Next to the ‘‘Go’’ button is a status indicator, it is not a button. Initially it is green and displays the word ‘‘OK’’. Pressing the ‘‘Go’’ button initiates the set operation. During that operation the status indicator turns blue. At the end of the operation the status indicator returns to green and displays the word ‘‘OK’’ if everything went all right. If something went wrong, the status indicator turns red and displays the words ‘‘NOT OK’’.

Recall that all BURT operations generate a log. To view this log, after performing an operation and receiving a ‘‘NOT OK’’ status for example, the user presses the ‘‘View Log ...’’ button. This will generate a new window containing the verbose version of BURT’s set operation log. The log may be printed using the ‘‘Print Log’’ button. A new log is generated each time the ‘‘Go’’ button is pressed, thus destroying the previous log.

At the end of the set operations, the user presses the ‘‘Done’’ button which removes the set window from the screen.

Chapter 6: Creating Your Own Snapshot Files

It was generally intended that users of BURT create their own request and dependency files and that BURT would create all the snapshot files. After all, BURT is a backup and restore tool, so it is not surprising to expect that a snapshot file that is being restored is the result of some previous backup, and therefore created by BURT. However, we recognize that this is not always the case and that it is possible to use BURT to write values, in the form of snapshot files, that were not the result of a previous backup, but rather the output from some program outside BURT's suite of programs. It is for this reason that we describe the steps necessary to facilitate those sophisticated users who wish to create their own snapshot files.

We provide this description with the following warning; users wishing to create their own snapshot files should only create SDDS snapshot files. Further, continued support for those users wishing to create their own snapshot files will come in the form of a C header file called *burtpublic.h* (explained below in section 1). ***Users who create their own non-SDDS snapshot files do so at their own risk.*** We provide no support for those users that choose to create their own non-SDDS snapshot files, and further, we reserve the right to change any or all of the syntax/format of non-SDDS snapshot files without any consideration for such users.

1. Getting Started - Including burtpublic.h

Users wishing to create their own snapshot files will be affected by any changes made in the format and/or syntax of those files, or changes in the names of the SDDS columns and parameters of those files. In order to minimize the users' grief caused by these changes, we made a portion of BURT's internal names available to the user community in the file *burtpublic.h,* which presumably can be found with the rest of BURT's source code. Contact your EPICS system administrator to find out where this file resides.

burtpublic.h contains all the column and parameter names as well as some other standard values all in the form of **#define** macros. Users wishing to create their own SDDS snapshot files are best shielded from any changes to these names and standard values by producing these files with a C program and using the **#include** facility to include *burtpublic.h*. The intention here is that, although we may change the column and parameter names found in SDDS snapshot files, it is unlikely that we will ever change the **#define**'d names.

2. Parameters

Snapshot files are comprised of two parts: the header section that identifies who took the snapshot, when the snapshot was taken, and what type of snapshot it is, and the data section where the process variable and device names along with their associated values appear. In SDDS snapshot files, the header section appears at the top of the file in the form of SDDS fixed parameters. Below is a table describing all the parameters BURT looks for in a snapshot file. **All parameters are strings**.

Public Names	Parameter Names	Required / Optional	Values	Default Values
TIMEHEADERSTRING	TimeStamp	optional	when snapshot was produced	1/1/70 00:00 GMT
LOGINHEADERSTRING	LoginId	optional	loginid and realworld name of user producing snapshot	UNKNOWNLOGINID (UNKNOWNREALWORLDNAME)
EFFUIDHEADERSTRING	EffectiveUID	optional	effective UID of user producing snapshot	UNKNOWN
GROUPIDHEADERSTRING	GroupID	optional	group ID of user producing snapshot	UNKNOWN
KEYWORDSHEADERSTRING	BurtKeywords	optional	user supplied keywords	empty string
COMMENTSHEADERSTRING	BurtComments	optional	user supplied comments	empty string
TYPEHEADERSTRING	SnapType	required	type of snapshot, ABSOLUTESTRING, RELATIVESTRING, or NOWRITESTRING	

Parameter Names are the names of the parameters as they currently (with respect to the creation of this document) appear. These names may change over time. **Name**s are the names of the parameters as they appear in *burtpublic.h*, names that are far less likely to change and users are therefore recommended to use these names instead of the ones found in **Parameter Names**.

All the parameters are strings and are defined as fixed values that are intended to pertain to the entire snapshot file. Note that all but one of the parameters are optional. The only required parameter is TYPEHEADERSTRING (SnapType). However, when supplying an optional parameter, the user must be very careful to format its value carefully. Examples of how to format each of the parameters correctly appears in the example in section 4.

The values of the parameters KEYWORDSHEADERSTING (BurtKeywords), COMMENTSHEADERSTRING (BurtComments), and TYPEHEADERSTRING (SnapType) are specified directly by the user. The values of the other parameters, TIMEHEADERSTRING (TimeStamp), LOGINHEADERSTRING (LoginID), EFFUIDHEADERSTRING (EffectiveUID), and GROUPIDHEADERSTRING (GroupID) are also supplied by the user, but only after making calls to operating system, i.e., UNIX. This is illustrated by the example in section 4.

3. Columns

The second portion of snapshot files is the data. In SDDS snapshot files the data appears as a single SDDS table. Below is a table describing the columns in that SDDS snapshot table.

Public Names	Column Names	SDDS Types	Required/ Optional	Contents	Default Values	Default Meanings
NAME_COL	ControlName	SDDS_STRING	required	PV or Device name		
TYPE_COL	ControlType	SDDS_STRING	required	type of entity, PVSTRING or DEVSTRING		
LINEAGE_COL	Lineage	SDDS_STRING	required	lineage of composite devices, for devices only, DEFAULT-STRING for pvars		
NELEM_COL	Count	SDDS_LONG	required	number of elements, 1 for devices		
VAL_COL	ValueString	SDDS_STRING	required	value		
MODE_COL	ControlMode	SDDS_STRING	optional	read only tag, DERAULTSTRING, READONLYSTRING, or READONLYNOTI-FYSTRING	DEFAULT-STRING	restore to IOC
READMSG_COL	BackupMsg	SDDS_STRING	optional	device read msg, DEFAULTSTRING for pvars	DEFAULT-STRING	DEFAULT-READMSG
WRITEMSG_COL	RestoreMsg	SDDS_STRING	optional	device write msg, DEFAULTSTRING for pvars	DEFAULT-STRING	DEFAULT-WRITEMSG

Similar to the parameter names, **Column Names** are the names of the columns as they currently appear in the SDDS snapshot files. These names may change over time. **Public Names** are the names of the columns as they appear in burtpublic.h, names that are far less likely to change and users are therefore recommended to use these names instead of the ones found in **Column Names**.

Note that all but three of the columns are required. Additionally, values of certain columns are dictated by values in other columns. For example, NELEM_COL (Count) must be 0 for devices and the columns LINEAGE_COL (Lineage), READMSG_COL (BackupMsg), and WRITEMSG_COL (RestoreMsg) must all be DEFAULT_STRING (currently defined as “-” in *burtpublic.h*) for process variables.

4. Example

In this section we present an example C program, *snap.c*, and that generates an SDDS snapshot file. In doing so, it does not leave out any of the optional columns or parameters. It points out which UNIX header files to include and how to retrieve and correctly format values from UNIX. It also illustrates how to correctly use the **#define**’d values from *burtpublic.h* to create the parameter and column names as well as how to define the lineage for devices and specify missing values in vectors. The program is presented with commentary. This program (without commentary) as well as its associated *makefile* and output file all appear as appendices to this document.

4.1. Include Files

```
/* for getpwuid() */
#include <pwd.h>

/* for time() */
#include <sys/types.h>
#include <sys/time.h>

/* for geteuid() */
#include <unistd.h>

#include <burtpublic.h>

#include <SDDS.h>
```

Initially we have the **#include** files. The first four are necessary for the subsequent UNIX calls we need to make to get information about the user generating the snapshot and the time at which the snapshot was generated. The last two are necessary for BURT and SDDS, respectively.

4.2. main()

```
main()
{

SDDS_TABLE table;
int effective_uid;
time_t currtime;
struct passwd *pp;
char buff[100];

    if (!SDDS_InitializeOutput(&table, SDDS_ASCII, 1L, NULL, NULL, "Snapshot"))
    {
        fprintf(stderr, "ERROR: unable to initialize table.\n");
        exit(1);
    } /* endif */
```

Above we have named the output file *Snapshot*. Below we define the snapshot header,i.e., the fixed valued parameters. Recall that BURT requires that only one of the parameters be defined, TYPEHEADERSTRING (SnapType). However, we have defined all for this program. First, we need to make some UNIX calls.

4.3. Necessary UNIX calls

```
    /***************************/
    /*                         */
    /* Defining the Parameters */
    /*                         */
    /***************************/

    /* first, make all the necessary UNIX calls */

    if (time(&currtime) == (time_t) -1)
    {
        fprintf(stderr, "ERROR: unable to get current time.\n");
        exit(1);
    } /* endif */

    effective_uid = geteuid();

    if ((pp = getpwuid(effective_uid)) == NULL)
    {
        fprintf(stderr, "ERROR: unable to get user information.\n");
```

```
    exit(1);
} /* endif */
```

Above are examples of the UNIX calls necessary identify who is creating the snapshot file and the time at which it is created. Following are the SDDS calls to create each of the fixed valued parameters. Note that the public names of the parameters, i.e., those found in *burtpublic.h*, are used and that each parameter has been defined as the correct SDDS type. Note also the format of the parameters, particularly the format of TIMEHEADERSTRING (TimeStamp) and LOGINHEADERSTING (LoginID). BURT expects and demands this exact syntax.

4.4. Fixed Value Parameters

```
/* time stamp */
if (!SDDS_DefineParameter(&table, TIMEHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, ctime(&currtime)) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        TIMEHEADERSTRING);
    exit(1);
} /* endif */

/* loginid (real world name) */
sprintf(buff, "%s (%s)", pp->pw_name, pp->pw_gecos);
if (!SDDS_DefineParameter(&table, LOGINHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, buff) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        LOGINHEADERSTRING);
    exit(1);
} /* endif */

/* effective uid */
/* uid - person who logged in ... effective uid person as */
/* defined by "set-user-ID" (if done at all)              */
sprintf(buff, "%d", (int) pp->pw_uid);
if (!SDDS_DefineParameter(&table, EFFUIDHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, buff) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        EFFUIDHEADERSTRING);
    exit(1);
} /* endif */

/* group id */
sprintf(buff, "%d", (int) pp->pw_gid);
if (!SDDS_DefineParameter(&table, GROUPIDHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, buff) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        GROUPIDHEADERSTRING);
    exit(1);
} /* endif */

/* keywords */
if (!SDDS_DefineParameter(&table, KEYWORDSHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, "these are keywords") == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        KEYWORDSHEADERSTRING);
    exit(1);
} /* endif */

/* comments */
if (!SDDS_DefineParameter(&table, COMMENTSHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, "these are comments") == -1)
{
```

```
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        COMMENTSHEADERSTRING);
    exit(1);
} /* endif */

/* snapshot type, Absolute */
if (!SDDS_DefineParameter(&table, TYPEHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, ABSOLUTESTRING) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        TYPEHEADERSTRING);
    exit(1);
} /* endif */
```

We have chosen to create an Absolute snapshot file by specifying the value ABSOLUTESTRING. We could have just as easily created a Relative or Nowrite snapshot file.

Immediately following is the definition of the columns. Although BURT does not require all the columns, we chose to include all of them. Note that each column has been defined with the correct public name and as the correct SDDS type.

4.5. Columns

```
/***********************/
/*                     */
/* Defining the Columns */
/*                     */
/***********************/

if (SDDS_DefineColumn(&table, NAME_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", NAME_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, TYPE_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", TYPE_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, LINEAGE_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", LINEAGE_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, READMSG_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", READMSG_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, WRITEMSG_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", WRITEMSG_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, MODE_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", MODE_COL);
    exit(1);
} /* endif */
```

```
if (SDDS_DefineColumn(&table, NELEM_COL,
    NULL, NULL, NULL, NULL, SDDS_LONG, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", NELEM_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, VAL_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", VAL_COL);
    exit(1);
} /* endif */
```

4.6. Writing header

```
/*********************/
/*                   */
/* Writing the Header */
/*                   */
/*********************/

if (!SDDS_WriteLayout(&table))
{
    fprintf(stderr, "ERROR: could not write header.\n");
    exit(1);
} /* endif */
```

4.7. Starting the table

```
/**************************/
/*                        */
/* Starting the Data Table */
/*                        */
/**************************/

if (!SDDS_StartTable(&table, 5L))
{
    fprintf(stderr, "ERROR: unable to start the data table\n");
    exit(1);
} /* endif */
```

Below we fill the table with data values. This particular table has five entries, two process variables and three devices.

4.8. Filling the table

```
/******************************/
/*                            */
/* Filling the Table With Data */
/*                            */
/******************************/
```

The first value is a scalar process variable, ‘‘burtgenerator’’, with a value of 0.0.

```
/* row 0: scalar pv, name="burtgenerator" val=0.0 */
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 0L,
    NAME_COL, "burtgenerator",
    TYPE_COL, PVSTRING,
    LINEAGE_COL, DEFAULTSTRING,
    READMSG_COL, DEFAULTSTRING,
    WRITEMSG_COL, DEFAULTSTRING,
    MODE_COL, DEFAULTSTRING,
    NELEM_COL, 1L,
    VAL_COL, "0.0",
```

```
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 0\n");
    exit(1);
} /* endif */
```

The second value is a device, ‘‘burtdevao’’, that is tagged Read Only and has a value of 1.0

```
/* row 1: device, name="burtdevao" val=1.0 mode=RO */
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 1L,
    NAME_COL, "burtdevao",
    TYPE_COL, DEVSTRING,
    LINEAGE_COL, DEFAULTSTRING,
    READMSG_COL, DEFAULTREADMSG,
    WRITEMSG_COL, DEFAULTWRITEMSG,
    MODE_COL, READONLYSTRING,
    NELEM_COL, 1L,
    VAL_COL, "1.0",
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 1\n");
    exit(1);
} /* endif */
```

The third value is also a device, ‘‘burtdevcalc’’, that is tagged Read Only Notify and has a value of 2.0.

```
/* row 2: device, name="burtdevcalc" val=2.0 mode=RON */
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 2L,
    NAME_COL, "burtdevcalc",
    TYPE_COL, DEVSTRING,
    LINEAGE_COL, DEFAULTSTRING,
    READMSG_COL, DEFAULTREADMSG,
    WRITEMSG_COL, DEFAULTWRITEMSG,
    MODE_COL, READONLYNOTIFYSTRING,
    NELEM_COL, 1L,
    VAL_COL, "2.0",
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 2\n");
    exit(1);
} /* endif */
```

The fourth value is a device, ‘‘burtdevcalc’’, that is part of a composite device called ‘‘burtdevcomp’’. Its value is 3.0 and it here we choose to explicitly supply its backup message, ‘‘read’’. Although its backup message happens to be the same as the DEFAULTREADMSG, we explicitly assigned the backup message its value to demonstrate how to do so. We could have just as easily chosen any backup message we wanted to.

```
/* row 3: device, name="burtdevcalc" val=3.0 parent=burtdevcomp */
/*                readmsg="read" */
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 3L,
    NAME_COL, "burtdevcalc",
    TYPE_COL, DEVSTRING,
    LINEAGE_COL, "burtdevcomp",
    READMSG_COL, "read",
    WRITEMSG_COL, DEFAULTWRITEMSG,
    MODE_COL, DEFAULTSTRING,
    NELEM_COL, 1L,
    VAL_COL, "3.0",
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 3\n");
    exit(1);
```

```
    } /* endif */
```

The fifth value is a vector process variable, ‘‘burtwaveform’’, that has six elements. The last two values of the vector are missing. Note when supplying values to a vector where some of the values are missing the user must use NULLSTRING (from *burtpublic.h*) to specify the missing values.

```
    /* row 4: vector pv, name="burtwaveform" nelem=6 */
    /*                   val=[4.0 4.1 4.2 4.3 4.4 <missing> <missing> ] */
    sprintf(buff, "4.0 4.1 4.2 4.3 %s %s", NULLSTRING, NULLSTRING);
    if (!SDDS_SetRowValues(&table,
        (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 4L,
        NAME_COL, "burtwaveform",
        TYPE_COL, PVSTRING,
        LINEAGE_COL, DEFAULTSTRING,
        READMSG_COL, DEFAULTSTRING,
        WRITEMSG_COL, DEFAULTSTRING,
        MODE_COL, DEFAULTSTRING,
        NELEM_COL, 6L,
        VAL_COL, buff,
        NULL))
    {
        fprintf(stderr, "ERROR: unable to set row 4\n");
        exit(1);
    } /* endif */
```

4.9. Writing the table

```
    /**************************/
    /*                        */
    /* Writing the Data Table */
    /*                        */
    /**************************/

    if (!SDDS_WriteTable(&table))
    {
        fprintf(stderr, "ERROR: could not write table.\n");
        exit(1);
    } /* endif */
```

4.10. Cleanup

```
    /***********/
    /*         */
    /* Cleanup */
    /*         */
    /***********/

    if (!SDDS_Terminate(&table))
    {
        fprintf(stderr, "ERROR: could not terminate SDDS table.\n");
        exit(1);
    } /* endif */

} /* end main() */
```

As mentioned, the previous program without interleaved commentary, its *makefile*, and the output it generates can all be found in the appendices to this document.

There is one more note of interest which has to do with reading SDDS snapshot files into programs outside the suite of BURT’s programs. As mentioned earlier, this document is not intended as an instructional aid for SDDS, however, there is one parameter in the SDDS snapshot files that merits further discussion, TIMEHEADERSTRING (TimeStamp). It has a format all its own and it is occasionally convenient to convert that string value to one of the UNIX conventions of the number of seconds since 00:00:00, January 1, 1970. This is done with the UNIX command **strptime()** which converts a string representation of time to the number of seconds. It expects, as one of its arguments, a description of the format of the string. This format can also be found in burtpublic.h in the **#define**'d variable TIMEFORMATSTRING.

Assuming that the value of the parameter has been read into a string pointed at by the variable `cp`, here is the code convert the string into one of UNIX's internal structures.

```
/* for strptime() */
#include <time.h>
#include <burtpublic.h>
main()
{
char *cp;
struct tm tmtime;
.
.
.
    strptime(cp, TIMEFORMATSTRING, &tmtime);
.
.
.
} /* end main() */
```

Chapter 7: Conclusion

1. Theory of Operation

Backing up and restoring sets of values in a dynamic database are simply read and write operations with respect to the database, respectively. Although the values used during restoration are typically the product of a previous back up, the two operations, reading and writing, are inherently independent, and so the following discussions of each are separate and independent.

Before starting our discussions of reading and writing, we need to understand the nature of databases. A database is a finite set of records. At any point in time there is a finite set of physical databases[1] active in the system. So, at a point in time, let *D* be the database defined by the union of all the physical databases active in the system.

Each record in *D* is a set of ordered attribute-value pairs where the attribute is the name of the field within the record and, as these databases are not in first normal form, the value is an ordered n-tuple of atomic data values (or simply values). Therefore, an ordered 3-tuple *<record name, field name, atom id>*, which we call a name, is needed to uniquely identify a value in *D*. For example, given a record *R* with field *F* where the value of *F* is an ordered pair, the name *<R, F, 2>* identifies the second value in that ordered pair. The process of reading/writing from/to a database is the process of reading/writing these values as identified by names.

The read operation is a binary function. It takes as input a database *D* and a name. It returns the value identified by name if name is in *D*, otherwise it returns ε, a null value to indicate that name was not in *D*.

The write operation is a ternary predicate. It takes as input a database *D*, a name, and the value to be written. If name is in *D*, the value is written there and returns true, otherwise it returns false.

1.A physical database is a finite non-empty set of records. For further explanation see [4].

Reading and writing sets of values are simply a matter of performing individual reads and writes using sets of appropriate input for each.

2. Limitations

As an implementation, BURT fails to meet the full functionality described by the theoretical framework in the previous section. That framework calls for the ability to read and write atomic data values as identified by a name. BURT cannot do that in the general sense. It is limited to reading and writing the first *n* values of a field. For example, BURT is not capable of reading only the second value from a field that contains more than one value. This inadequacy has also manifested itself in **ca_get()**'s and **ss_get()**'s limitations.

Appendix A: Command Usage

1. burtconvertsnap

Usage:

burtconvertsnap snap {-l logfile} {-o outfile} {-v} {-sdds or -nosdds}

Where:

snap - Snapshot filename. This is the only switch that is not optional. You must specify exactly one snapshot file. This is the snapshot file that will be converted.

-l logfile - Log filename. The name of the file where all logging messages (e.g. error messages) go. The default is *stderr*.

-o outfile - Output snapshot filename. This is where the newly generated converted snapshot file will be placed. The default is *stdout*.

-v - Verbose. This increases the amount of information displayed in the logfile.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to produce whatever the input snapshot file is not.

2. burtmath

Usage:

burtmath (snap1 snap2 {-add or -sub}) or (snap1 -mult m) {-l logfile} {-o outfile} {-v} {-c ... comments ...} {-k keyword1 ... keywordn} {-sdds or -nosdds}

Where:

snap1 snap2 {-add or -sub} - Adding/subtracting snapshots. Here you wish to either add or subtract exactly two snapshot files. You must specify exactly two snapshot files when performing. either addition or subtraction. The default operation is -sub (subtraction).

snap1 -mult m - Multiplying snapshots. Here you wish to multiply all the values in snap1 by multiplication factor m.

-l logfile - Log file name. The name of the file where all logging messages (e.g. error messages, reports of process variables that were not found) go. The default is *stderr.*

-o outfile - Result file name. The name of the file where the resulting information goes. The default is *stdout*.

-v - Verbose. This increases the amount of information displayed in the logfile.

-c ... comments ... - Comments. Adds comments to the header of the result file.

-k keyword1 ... keywordn - Keywords. Adds keywords to the header of the result file.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to adopt the SDDS type from the input(s). If there is a heterogenous set of inputs (some SDDS and some non-SDDS), the default is to produce and SDDS compliant snapshot file.

3. burtrb

Usage:

burtrb -f req1 {req2 ...} {-l logfile} {-o outfile} {-d} {-v} {-c ... comments ...} {-k keyword1 ... keywordn} {-r retry count} {-sdds or -nosdds}

Where:

-f req1 {req2 ...} - Request file names. This is the only switch that is not optional. You must specify at least one request file.

-l logfile - Log file name. The name of the file where all logging messages (e.g. error messages, reports of process variables that were not found) go. The default is *stderr*.

-o outfile - Snapshot file name. The name of the file where the snapshot information goes. The default is *stdout*.

-d - Debug. Save the files created by processing the request files with the C preprocessor. The default is to delete these files.

-v - Verbose. This increases the amount of information displayed in the logfile.

-c ...comments ... - Comments. Adds comments to the header of the snapshot file.

-k keyword1 ... keywordn - Keywords. Adds keywords to the header of the snapshot file.

-r retry count - Number of additional attempts to wait for connections. The program will attempt to find all the process variables. If it is unsuccessful, it will try this many more times to establish connections. The default value is 0.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to adopt the SDDS type from the input(s). If there is a heterogenous set of inputs (some SDDS and some non-SDDS), the default is to produce an SDDS compliant snapshot file.

4. burtset

Usage:

burtset -f req1 req2 {-union or -inter or -diff} {-l logfile} {-o outfile} {-d} {-v} {-sdds or -nosdds}

Where:

-f req1 req2 - Request filenames. This is the only switch that is not optional. You must specify at least two request files.

-union or -inter or -diff - Set operation. The set operation (union, intersection, or difference) to be performed on the two request files. The default is union.

-l logfile - Log file name. The name of the file where all logging messages (e.g. error messages) go. The default is *stderr*.

-o outfile - Request file name. The name of the file where the result of the set operation goes. The default is *stdout*.

-d - Debug. Save the files created by processing the request files with the C preprocessor. The default is to delete these files.

-v - Verbose. This increases the amount of information displayed in the logfile.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to adopt the SDDS type from the inputs. If there is a heterogenous set of inputs (one SDDS and the other non-SDDS), the default is to produce and SDDS compliant snapshot file.

5. burtwb

Usage:

burtwb -f snap1 {snap2 ...} {-l logfile} {-o outfile} {-c ... comments ...}
{-k keyword1 ... keywordn} {-d} {-v} {-p dep1 ... depn} {-r retry count}
{-add} {-replace} {-sdds or -nosdds}

Where:

-f snap1 {snap2 ...} - Snapshot file names. This is the only switch that is not optional. You must specify at least one snapshot file.

-l logfile - Log file name. The name of the file where all logging messages (e.g. error messages, reports of process variables that were not found) go. The default is *stderr*.

-o outfile - Snapshot file name. If any of the snapshot files Read Only Notify values, this file is created and those values are placed there. If none of the snapshot files have Read Only Notify values, then no file is created. The default is *stdout*.

-c ... comments ... - Comments. Adds comments to the header of the snapshot file.

-k keyword1 ... keywordn - Keywords. Adds keywords to the header of the snapshot file.

-d - Debug. Save the files created by processing the dependency files with the C preprocessor. The default is to delete these files.

-v - Verbose. This increases the amount of information displayed in the logfile.

-p dep1 ... depn - Dependency file names. The names of the dependency files containing predicates (Boolean conditions) to be evaluated before writing the values from the snapshot files.

-r retry count - Number of additional attempts to wait for connections. The program will attempt to find all the process variables. If it is unsuccessful, it will try this many more times to establish connections. The default value is 0.

-add - Absolute snapshots written as adds. All the absolute snapshots, i.e., those taken directly off IOCs, will be written as additions to the values found on the IOCs. The default is to write the absolute snapshots as replacement values on the IOCs.

-replace - Relative snapshots written as replacements. All the relative snapshots, i.e., those generated by adding or subtracting two snapshots, will be written to replace the values on the IOCs. The default is to write the relative snapshots as additions to the values on the IOCs.

-sdds or -nosdds - SDDS/non-SDDS snapshot file. Explicitly specifying that the generated snapshot file will be SDDS/non-SDDS compliant. The default is to adopt the SDDS type from the input(s). If there is a heterogenous set of inputs (some SDDS and some non-SDDS), the default is to produce an SDDS compliant snapshot file.

Appendix B: BURT Header File

1. burtpublic.h

```
#ifndef __burtpublic__
#define __burtpublic__

#define NULLSTRING "\\0"
#define TIMEFORMATSTRING "%a %h %d %T %Y" /* "man stpftime" to understand */
                      /* format of output from ctime() */

/* Start SDDS Snapshot Parameters */
#define TIMEHEADERSTRING     "TimeStamp"
#define LOGINHEADERSTRING    "LoginID"
#define EFFUIDHEADERSTRING   "EffectiveUID"
#define GROUPIDHEADERSTRING  "GroupID"
#define KEYWORDSHEADERSTRING "BurtKeywords"
#define COMMENTSHEADERSTRING "BurtComments"
#define TYPEHEADERSTRING     "SnapType"
/* End SDDS Snapshot Parameters */

#define READONLYSTRING       "RO"
#define READONLYNOTIFYSTRING "RON"

#define ABSOLUTESTRING "Absolute"
#define RELATIVESTRING "Relative"
#define NOWRITESTRING  "Nowrite"

/* Start SDDS */
/* column names */
#define NAME_COL     "ControlName"
#define TYPE_COL     "ControlType"
#define LINEAGE_COL  "Lineage"
#define READMSG_COL  "BackupMsg"
#define WRITEMSG_COL "RestoreMsg"
#define NELEM_COL    "Count"
#define MODE_COL     "ControlMode"
#define VAL_COL      "ValueString"
```

```
/* string constants found in req files */
#define PVSTRING  "pv"
#define DEVSTRING "dev"

/* default messages */
#define DEFAULTREADMSG  "read"
#define DEFAULTWRITEMSG "set"

#define DEFAULTSTRING    "-"
#define LINEAGEDELIMETER ","

/* End SDDS */

#define UNKNOWN              -1
#define UNKNOWNLOGINID       "Unknown"
#define UNKNOWNREALWORLDNAME "Unknown"

#endif
```

Appendix C: Sample BURT Program

1. snap.c

```
/* for getpwuid() */
#include <pwd.h>

/* for time() */
#include <sys/types.h>
#include <sys/time.h>

/* for geteuid() */
#include <unistd.h>

#include <burtpublic.h>

#include <SDDS.h>

main()
{

SDDS_TABLE table;
int effective_uid;
time_t currtime;
struct passwd *pp;
char buff[100];

    if (!SDDS_InitializeOutput(&table, SDDS_ASCII, 1L, NULL, NULL, "Snapshot"))
    {
        fprintf(stderr, "ERROR: unable to initialize table.\n");
        exit(1);
    } /* endif */

    /***************************/
    /*                         */
    /* Defining the Parameters */
    /*                         */
    /***************************/
```

```
/* first, make all the necessary UNIX calls */

if (time(&currtime) == (time_t) -1)
{
    fprintf(stderr, "ERROR: unable to get current time.\n");
    exit(1);
} /* endif */

effective_uid = geteuid();

if ((pp = getpwuid(effective_uid)) == NULL)
{
    fprintf(stderr, "ERROR: unable to get user information.\n");
    exit(1);
} /* endif */

/* time stamp */
if (!SDDS_DefineParameter(&table, TIMEHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, ctime(&currtime)) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        TIMEHEADERSTRING);
    exit(1);
} /* endif */

/* loginid (real world name) */
sprintf(buff, "%s (%s)", pp->pw_name, pp->pw_gecos);
if (!SDDS_DefineParameter(&table, LOGINHEADERSTING,
    NULL, NULL, NULL, NULL, SDDS_STRING, buff) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        LOGINHEADERSTING);
    exit(1);
} /* endif */

/* effective uid */
/* uid - person who logged in ... effective uid person as */
/* defined by "set-user-ID" (if done at all)              */
sprintf(buff, "%d", (int) pp->pw_uid);
if (!SDDS_DefineParameter(&table, EFFUIDHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, buff) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        EFFUIDHEADERSTRING);
    exit(1);
} /* endif */

/* group id */
sprintf(buff, "%d", (int) pp->pw_gid);
if (!SDDS_DefineParameter(&table, GROUPIDHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, buff) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        GROUPIDHEADERSTRING);
    exit(1);
} /* endif */

/* keywords */
if (!SDDS_DefineParameter(&table, KEYWORDSHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, "these are keywords") == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        KEYWORDSHEADERSTRING);
    exit(1);
} /* endif */

/* comments */
```

```
if (!SDDS_DefineParameter(&table, COMMENTSHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, "these are comments") == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        COMMENTSHEADERSTRING);
    exit(1);
} /* endif */

/* snapshot type, Absolute */
if (!SDDS_DefineParameter(&table, TYPEHEADERSTRING,
    NULL, NULL, NULL, NULL, SDDS_STRING, ABSOLUTESTRING) == -1)
{
    fprintf(stderr, "ERROR: could not define parameter >%s<.\n",
        TYPEHEADERSTRING);
    exit(1);
} /* endif */

/***********************/
/*                     */
/* Defining the Columns */
/*                     */
/***********************/

if (SDDS_DefineColumn(&table, NAME_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", NAME_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, TYPE_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", TYPE_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, LINEAGE_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", LINEAGE_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, READMSG_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", READMSG_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, WRITEMSG_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", WRITEMSG_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, MODE_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", MODE_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, NELEM_COL,
    NULL, NULL, NULL, NULL, SDDS_LONG, 0) == -1)
{
    fprintf(stderr, "ERROR: could not define column >%s<.\n", NELEM_COL);
    exit(1);
} /* endif */
if (SDDS_DefineColumn(&table, VAL_COL,
    NULL, NULL, NULL, NULL, SDDS_STRING, 0) == -1)
{
```

```
    fprintf(stderr, "ERROR: could not define column >%s<.\n", VAL_COL);
    exit(1);
} /* endif */

/*********************/
/*                   */
/* Writing the Header */
/*                   */
/*********************/

if (!SDDS_WriteLayout(&table))
{
    fprintf(stderr, "ERROR: could not write header.\n");
    exit(1);
} /* endif */

/**************************/
/*                        */
/* Starting the Data Table */
/*                        */
/**************************/

if (!SDDS_StartTable(&table, 5L))
{
    fprintf(stderr, "ERROR: unable to start the data table\n");
    exit(1);
} /* endif */

/******************************/
/*                            */
/* Filling the Table With Data */
/*                            */
/******************************/

/* row 0: scalar pv, name="burtgenerator" val=0.0 */
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 0L,
    NAME_COL, "burtgenerator",
    TYPE_COL, PVSTRING,
    LINEAGE_COL, DEFAULTSTRING,
    READMSG_COL, DEFAULTSTRING,
    WRITEMSG_COL, DEFAULTSTRING,
    MODE_COL, DEFAULTSTRING,
    NELEM_COL, 1L,
    VAL_COL, "0.0",
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 0\n");
    exit(1);
} /* endif */

/* row 1: device, name="burtdevao" val=1.0 mode=RO */
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 1L,
    NAME_COL, "burtdevao",
    TYPE_COL, DEVSTRING,
    LINEAGE_COL, DEFAULTSTRING,
    READMSG_COL, DEFAULTREADMSG,
    WRITEMSG_COL, DEFAULTWRITEMSG,
    MODE_COL, READONLYSTRING,
    NELEM_COL, 1L,
    VAL_COL, "1.0",
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 1\n");
    exit(1);
} /* endif */
```

```
/* row 2: device, name="burtdevcalc" val=2.0 mode=RON */
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 2L,
    NAME_COL, "burtdevcalc",
    TYPE_COL, DEVSTRING,
    LINEAGE_COL, DEFAULTSTRING,
    READMSG_COL, DEFAULTREADMSG,
    WRITEMSG_COL, DEFAULTWRITEMSG,
    MODE_COL, READONLYNOTIFYSTRING,
    NELEM_COL, 1L,
    VAL_COL, "2.0",
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 2\n");
    exit(1);
} /* endif */

/* row 3: device, name="burtdevcalc" val=3.0 parent=burtdevcomp */
/*                 readmsg="read" */
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 3L,
    NAME_COL, "burtdevcalc",
    TYPE_COL, DEVSTRING,
    LINEAGE_COL, "burtdevcomp",
    READMSG_COL, "read",
    WRITEMSG_COL, DEFAULTWRITEMSG,
    MODE_COL, DEFAULTSTRING,
    NELEM_COL, 1L,
    VAL_COL, "3.0",
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 3\n");
    exit(1);
} /* endif */

/* row 4: vector pv, name="burtwaveform" nelem=6 */
/*                    val=[4.0 4.1 4.2 4.3 4.4 <missing> <missing> ] */
sprintf(buff, "4.0 4.1 4.2 4.3 %s %s", NULLSTRING, NULLSTRING);
if (!SDDS_SetRowValues(&table,
    (long) (SDDS_SET_BY_NAME | SDDS_PASS_BY_VALUE), 4L,
    NAME_COL, "burtwaveform",
    TYPE_COL, PVSTRING,
    LINEAGE_COL, DEFAULTSTRING,
    READMSG_COL, DEFAULTSTRING,
    WRITEMSG_COL, DEFAULTSTRING,
    MODE_COL, DEFAULTSTRING,
    NELEM_COL, 6L,
    VAL_COL, buff,
    NULL))
{
    fprintf(stderr, "ERROR: unable to set row 4\n");
    exit(1);
} /* endif */

/*************************/
/*                       */
/* Writing the Data Table */
/*                       */
/*************************/

if (!SDDS_WriteTable(&table))
{
    fprintf(stderr, "ERROR: could not write table.\n");
    exit(1);
} /* endif */
```

```
    /***********/
    /*         */
    /* Cleanup */
    /*         */
    /***********/

    if (!SDDS_Terminate(&table))
    {
        fprintf(stderr, "ERROR: could not terminate SDDS table.\n");
        exit(1);
    } /* endif */

} /* end main() */
```

2. makefile

```
CC = acc

EPICS = /net/phebos/epics/add_on

BURTINCDIR = -I$(EPICS)/src/burt
SDDSINCDIR = -I$(EPICS)/src/sdds/include

INCLUDEDIRS = $(BURTINCDIR) $(SDDSINCDIR)

SDDSLIBDIR = -L$(EPICS)/lib

LIBDIRS = $(SDDSLIBDIR)

LIBS = -lSDDS1 -lnamelist -lmdblib
CFLAGS = -c

snap:snap.o
    $(CC) -o snap snap.o $(LIBDIRS) $(LIBS)

clean:
    /bin/rm -f *.o snap

.c.o:$*.c
    $(CC) $(CFLAGS) $(INCLUDEDIRS) $*.c
```

Appendix D: SDDS Snapshot File

```
SDDS1
&description &end
&parameter
 name = “TimeStamp”,  type = “string”,  fixed_value = “Mon Jul 25 11:01:42 1994
“, &end
&parameter
 name = “LoginID”,  type = “string”,  fixed_value = “karonis (Nicholas T. Karonis)”,
&end
&parameter
 name = “EffectiveUID”,  type = “string”,  fixed_value = “164”, &end
&parameter
 name = “GroupID”,  type = “string”,  fixed_value = “55”, &end
&parameter
 name = “BurtKeywords”,  type = “string”,  fixed_value = “these are keywords”, &end
&parameter
 name = “BurtComments”,  type = “string”,  fixed_value = “these are comments”, &end
&parameter
 name = “SnapType”,  type = “string”,  fixed_value = “Absolute”, &end
&column
 name = “ControlName”,  type = “string”, &end
&column
 name = “ControlType”,  type = “string”, &end
&column
 name = “Lineage”,  type = “string”, &end
&column
 name = “BackupMsg”,  type = “string”, &end
&column
 name = “RestoreMsg”,  type = “string”, &end
&column
 name = “ControlMode”,  type = “string”, &end
&column
 name = “Count”,  type = “long”, &end
&column
 name = “ValueString”,  type = “string”, &end
&data
 mode = “ascii”, &end
! table number 1
5                                       ! number of rows
```

```
burtgenerator pv - - - - 1 0.0
burtdevao dev - read set RO 1 1.0
burtdevcalc dev - read set RON 1 2.0
burtdevcalc dev burtdevcomp read set - 1 3.0
burtwaveform pv - - - - 6 "4.0 4.1 4.2 4.3 \0 \0"
```

Bibliography

[1] M. Borland, ‘‘Application Programmer’s Guide for SDDS Version 1’’, Argonne National Laboratory, Advanced Photon Source, December 1993.

[2] R. Cole, ‘‘Archiving Reference Manual’’, Los Alamos National Laboratory, January 1993.

[3] J.O. Hill, ‘‘Channel Access Reference Manual’’, Los Alamos National Laboratory.

[4] N.T. Karonis and M.R. Kraimer, ‘‘Links in a Distributed Database: Theory and Implementation’’, Argonne National Laboratory, Advanced Photon Source, December 1991.

[5] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1978.

[6] M. R. Kraimer, ‘‘Experimental Physics Industrial Control System (EPICS) Input/Output Controller (IOC) Application Developer’s Guide’’, Argonne National Laboratory, Advanced Photon Source, May 1994.

[7] P.J. Plauger, *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J., 1992.

[8] C. Saunders, ‘‘Device Access Library: User’s Guide and Reference’’, Argonne National Laboratory, Advanced Photon Source.