The Pablo Self-Defining Data Format

Ruth A. Aydt*

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

March 17, 1992
Revised February 15, 1993
Revised May 6, 1993
Revised April 11, 1994

Copyright © 1992-1994 The University of Illinois Board of Trustees. All Rights Reserved.
Pablo is a registered trademark of the Board of Trustees of the University of Illinois.

*Supported in part by the Advanced Research Projects Agency under ARPA Contract Number DABT63-91-K-0004, by the National Science Foundation under grants NSF CCR87-06653 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, by a grant from the Digital Equipment Corporation External Research Program, and by a collaborative research agreement with the Intel Supercomputer Systems Division.

Contents

Preface ii

1 Introduction 1
- 1.1 Motivation for the Development of SDDF . . . 1
- 1.2 The Pablo Self-Defining Data Format . . . 2
- 1.3 The SDDF Interface Library . . . 3
- 1.4 Using the Self-Defining Data Format . . . 3

2 SDDF Basics—Explanation by Example 4
- 2.1 A Sample SDDF File . . . 4
- 2.2 A Sample Program using the SDDF Interface Library . . . 4

3 The Components of an SDDF File 9
- 3.1 SDDF File Headers . . . 9
- 3.2 SDDF Packet Types . . . 9
- 3.3 Command Packets . . . 10
- 3.4 Stream Attribute Packets . . . 10
- 3.5 Record Descriptor Packets . . . 11
- 3.6 Record Data Packets . . . 13

4 The SDDF Interface Library 15
- 4.1 InputFileStreamPipe and OutputFileStreamPipe . . . 15
- 4.2 PipeReader . . . 15
- 4.3 AsciiPipeWriter, BinaryPipeWriter, and ConversionPipeWriter . . . 16
- 4.4 CString . . . 18
- 4.5 Attributes . . . 18
- 4.6 FieldDescriptor . . . 18
- 4.7 StructureDescriptor . . . 18
- 4.8 Value . . . 19
- 4.9 Array . . . 20
- 4.10 RecordDossier . . . 21
- 4.11 RecordDictionary . . . 21

A Grammar for SDDF File - ASCII Representation 23
- A.1 Data File Production Rules . . . 23
- A.2 Stream Attribute Production Rules . . . 23
- A.3 Record Descriptor Production Rules . . . 23
- A.4 Record Data Production Rules . . . 23
- A.5 Command Production Rule . . . 24
- A.6 Miscellaneous Production Rules . . . 24

B Grammar for SDDF File - Binary Representation **25**
B.1 Data File Production Rule 25
B.2 Binary Header Production Rules 25
B.3 Packet Production Rules 26
B.4 Stream Attribute Production Rule 26
B.5 Record Descriptor Production Rules 26
B.6 Record Data Production Rules 26
B.7 Command Production Rule 27
B.8 Attribute List Production Rules 27
B.9 Miscellaneous Production Rules 27
B.10 Integer Constants 27
B.11 Numeric Constants Represented as Bytes 28

C Program to Convert SDDF File to Binary Representation **29**

D Program to Generate a Sample SDDF File **31**

E Sample SDDF File—ASCII Representation **37**

F Revision History **40**

References **41**

Preface

As parallel systems gain wider acceptance, the lack of tools for evaluating the performance of application and system software running on these machines becomes increasingly obvious and painful. To address this deficiency, the Picasso group at the University of Illinois is developing an integrated performance analysis environment—*Pablo*—that provides for the collection, transformation, and presentation of performance data across both SIMD and MIMD distributed memory parallel systems. Critical to the Pablo environment's goals of *scalability*, *portability*, and *extensibility*, are the event trace records that encapsulate the collected performance information and present it to the analysis component of the system for evaluation. Because there is no existing standard format for storing performance data, we developed the Pablo Self-Defining Data Format to represent the trace event information we capture and analyze.

The Pablo Self-Defining Data Format is a flexible file meta-format that specifies both data record structures and data record instances. By virtue of describing arbitrary data records, not just data records having pre-defined layouts, this meta-format allows new trace event types to be added with ease. The Pablo Self-Defining Data Format provides a syntax whereby scalar and array fields of base types (character, integer, single and double precision floating point) are grouped together to define new data record types—similar to declaring a structure in the C programming language. Data record instances contain the performance trace event information gathered on the parallel system being studied. Each data record instance includes an identifier linking it to a previously defined record structure that is used to interpret the information in the record.

The Pablo Self-Defining Data Format, by virtue of being portable and extensible, has application outside the scope of the Pablo performance evaluation environment. This manual documents the Pablo Self-Defining Data Format and presents it as a format for recording performance trace event data in general.

This work was funded by DARPA Contract No. DABT63–91–K–0004, National Science Foundation grants NSF CCR86–57696, NSF CCR87–06653 and NSF CDA87–22836 (Tapestry), NASA ICLASS Contract No. NAG–1–613 and by grants from the Digital Equipment Corporation External Research Program.

Ruth A. Aydt
March 17, 1992

1 Introduction

This manual documents the Pablo Self-Defining Data Format (SDDF), a flexible file meta-format designed to describe the structural layout of event records in performance trace files. We present our motivation for developing SDDF, high-level and in-depth coverage of the format itself, an explanation of the C++ interface library, and sample files and code demonstrating the use of SDDF.

1.1 Motivation for the Development of SDDF

The Picasso group at the University of Illinois is developing an integrated performance data analysis environment called Pablo [2]. The goal of the Pablo system is to provide tools for the collection and analysis of performance data across both SIMD and MIMD distributed memory parallel systems. The data capture component of the environment produces files containing performance trace event records for the system being studied. The data analysis component allows the user to manipulate and display the information contained in the trace files to gain insights into the behavior of the parallel system being studied. The format of the performance trace files should promote the general Pablo philosophies of *portability*, *scalability*, and *extensibility*.

While many scientific communities have well-established data formats for storing and distributing information critical to their fields, no such standard exists to specify the format of performance trace event records. The norm has been for each developer to define a fixed record format capable of representing those events of interest on the particular system being studied. While this approach addresses immediate needs, it leaves us without a consistent trace file format for representing performance data across a variety of machines. This lack of a standard "data language" makes it difficult to share results, and impossible to use analysis tools developed on one system to study traces collected on another. Without a pre-existing standard to guide us, we set out to define a trace file format which would satisfy the minimal requirements dictated by the nature of the data we are processing:

- Trace files are often quite large—potentially megabytes or even gigabytes in size—hence compactness is critical and data compression should not be precluded.
- Traces may be collected on one machine and analyzed on another; consequently the format must be able to accommodate potential differences in byte-order, word length, and floating point representation.
- The "interesting" performance data will vary based on the system component being studied (application software, system software, hardware) as well as on underlying architecture-specific characteristics (shared or distributed memory). Therefore, a diverse set of event types must be provided.
- The trace format should accommodate not only those event types currently recognized as valuable, but also allow new types to be added as needed to enhance the performance tuning effort.

To summarize: the format for performance trace event files must be *compact*, *portable*, *general*, and *extensible*. These requirements could not be addressed satisfactorily by adopting yet another fixed record format, so we investigated possible alternatives.

Previous forays by other researchers into self-defining data files such as HDF [1] and netCDF [3] have been very successful, and it seemed a similar approach might be well suited to the needs of the Pablo system. The scientific and graphical data sets targeted by HDF and netCDF usually consist of a small number of large records, often containing arrays of floating point data. In contrast, a performance trace data file usually consists of many event types with potentially thousands of data records of each type. The event records are often quite short, with integers and character strings appearing as frequently as single and double precision floating point values, and scalar values more common than arrays. To accommodate the distinctive nature of performance trace data, we developed a new self-defining data format.

1.2 The Pablo Self-Defining Data Format

The Pablo Self-Defining Data Format (SDDF) is a trace description language that specifies both data record structures and data record instances. Because the format can describe arbitrary data records, it is best viewed as a data *meta-format*.

The meta-format approach does not prescribe a fixed set of supported event types, the information that should be included for a given type, or how a specific data type (such as an integer) will be stored. That is, data semantics are not dictated by the trace file format. Instead, the collector of the trace data is free decide what is appropriate for each situation and describe event records accordingly. Correspondingly, the analyzer of the data can use the embedded descriptions to interpret the data.

To reiterate, a valid SDDF file is one that follows the SDDF meta-format syntax; there is no attempt to impose semantic meaning or correctness on the contents of the SDDF file. [1] This meta-format approach addresses the requirements of generality and extensibility by making it easy to have a diverse set of trace event types and simple to add new event types as they are needed.

The desire for compactness is best satisfied by a binary trace format, but byte order and number representation disparity across machines makes binary files inherently non-portable. ASCII files have the advantage of portability and are human-readable, but fail the test of compactness. Rather than attempting to choose the lesser of two evils, we developed SDDF grammars for both binary and ASCII representations. The binary format can be used when compactness is of utmost concern, and the ASCII format provides portability when needed. [2] Software tools are provided to convert SDDF trace files from one representation to the other.

To benefit as much as possible from the compactness of the binary format, the binary SDDF grammar supports a limited degree of portability across machines with different byte order conventions. Every binary SDDF file contains a flag indicating the byte ordering used in the file. If the byte order of a binary SDDF file does not match the byte order of the machine processing the file, the bytes of each basic data type are swapped automatically as the file is read. Although the size of a binary SDDF file is not dependent on the byte order, a consideration when electing to use a "reverse byte ordered" binary file is the increased processing time required to perform the byte swapping operations. The ASCII representation must be employed to share SDDF files between machines with differences such as word length or floating point representation.

[1] Note that nothing in the definition of SDDF restricts its usage to the encoding of performance trace data.

[2] In our experience, binary SDDF files have been anywhere from 42% to 75% smaller than the corresponding ASCII files.

1.3 The SDDF Interface Library

In addition to the meta-format itself, we developed a library of C++ classes that provides an interface to the data stored in SDDF files. The classes and their associated methods support the full range of operations necessary to write files in the SDDF format and to interpret the data stored in SDDF files. Using the interface library to produce an SDDF file, one first creates new class instances describing the desired event types and then populates data records of those types using C++ class methods provided. When reading an SDDF file, routines exist to recognize and save the event type descriptions; information contained in data record instances is then interpreted by referring to the appropriate type description. Particular data values are accessed via the record and field names assigned in the event type description—it is never necessary to directly manipulate bytes in the SDDF file.

1.4 Using the Self-Defining Data Format

There are actually several mechanisms for creating SDDF files; the only constraint is that the resulting file conform to the SDDF syntax rules. As outlined in the previous section, one mechanism for creating an SDDF file is to manipulate data structures and write records entirely through the interface provided by the C++ class library. By selecting this approach, the programmer is assured that the syntax will be correct and need only worry about describing the event types accurately and populating the event records with valid data. At the other end of the spectrum, a person can write code to produce ASCII or binary SDDF records directly, following the specifications of the appropriate SDDF grammar. This method forces the programmer to pay careful attention to the SDDF syntax for each record produced, but does allow the source code to be written in any language. It is also possible to write SDDF files using a hybrid of these two extremes—there is no single correct approach.

This manual documents SDDF at all levels and prepares the reader to make wise decisions on how the meta-format and the class library can best satisfy their particular requirements. Chapter 2 highlights the flexibility provided by the SDDF meta-format and introduces the interface library through the presentation of a sample SDDF file and the program used to generate the file. Chapter 3 describes in detail the components of an SDDF file, while Chapter 4 provides an in-depth look at the classes and methods of the interface library. The SDDF grammars and sample programs further demonstrating the library functionality can be found in the Appendices.

2 SDDF Basics—Explanation by Example

The basics of the SDDF meta-format can perhaps best be described by example. In this chapter, a short ASCII SDDF file demonstrates how data record structures and data record instances are represented using SDDF. The code listing for the program that generated the sample file provides a brief introduction to the classes and methods available in the C++ interface library.

2.1 A Sample SDDF File

The SDDF meta-format describes four distinct classes of records, each serving a specific purpose in the context of a complete SDDF file.

- Command: conveys action to be taken
- Stream Attribute: gives information pertinent to the entire file
- Record Descriptor: declares record structure
- Record Data: encapsulates data values

In the context of performance trace event records, a Stream Attribute might contain the date the trace was collected, a Record Descriptor would be included for each trace event type, and every Record Data instance would contain the data values for a single trace event.

Figure 1 shows a simple ASCII SDDF file containing performance trace data. The file contains a single Stream Attribute, two Record Descriptors, and four Record Data instances. The Stream Attribute provides general information about the trace file: it was created February 1, 1992. The Record Descriptors define two distinct event types: **`message send`** and **`context switch`**. The lines "#1:" and "#2:" associate unique tags with each of the event types. Each Record Data instance contains data values relating to a single captured trace event. The file includes performance trace information for two **`message send`** events and two **`context switch`** events.

Using the SDDF meta-format, one defines new event types by including the appropriate Record Descriptors in the file. SDDF does not assign semantic meaning to the data—the data values in the Record Data instances are interpreted by referring to the corresponding Record Descriptor. In the ASCII representation, the identifier used to choose the correct Record Descriptor is the record name (**`message send`** or **`context switch`**). [3] By looking at the **`context switch`** Record Descriptor and the first Record Data instance, we see that a context switch to the file i/o state occurred at time 100.15 on processor two. The second Record Data instance can be interpreted using the **`message send`** Record Descriptor: processor zero sent a 512 byte message at time 101.1 to processors one, three, five, and seven.

The components of SDDF files and the syntax rules of the meta-format are described in detail in Chapter 3.

2.2 A Sample Program using the SDDF Interface Library

The interface library provides a set of C++ classes to aid in processing SDDF files. Through library routines, the programmer can use field names assigned in Record Descriptors to populate Record

[3] In the binary representation, the unique tag is used as the identifier.

```
SDDFA
/*
 * "run date" "February 1, 1992"
 */ ;;

#1:
// "event" "message sent to one or more processors"
"message send" {
        double  "timestamp";
        // "Source" "Processor Sending Message"
        int     "sourcePE";
        // "Destination" "Processor(s) Receiving Message"
        int     "destinationPE"[];
        // "Size" "Message length in bytes"
        int     "messageLength";
};;

#2:
"context switch" {
        double  "timestamp";
        int     "processor";
        char    "processName"[];
};;

"context switch" { 100.150000, 2,
        [30] {
        "file i/o"
        } };;

"message send" { 101.100000, 0,
        [4] {
        1,
        3,
        5,
        7
        }, 512 };;

"message send" { 102.150000, 7,
        [1] {
        1
        }, 1012 };;

"context switch" { 108.000000, 4,
        [30] {
        "idle"
        } };;
```

Figure 1: Sample SDDF file—ASCII representation.

```
 1 #include "AsciiPipeWriter.h"
 2 #include "Attributes.h"
 3 #include "OutputFileStreamPipe.h"
 4 #include "RecordDossier.h"
 5 #include "StructureDescriptor.h"
 6 #include "InitializeStatic.C"

 7 main()
 8 {
 9     Attributes              attributes;
10     StructureDescriptor     *structureP;
11     FieldDescriptor         *fieldP;

12     /** Open file **/
13     OutputFileStreamPipe *outFile = new OutputFileStreamPipe( "MyFile" );
14     PipeWriter *pipeWriter = new AsciiPipeWriter( outFile );

15     /** Stream Attribute **/
16     attributes.insert( "run date", "February 1, 1992" );
17     pipeWriter->putAttributes( attributes );
```

Figure 2: Program to generate sample SDDF file—part 1 of 3 parts.

Data instances with data values. The library interface makes it easy to generate syntactically correct files and hides differences between the ASCII and binary SDDF representations; the method calls are the same regardless of the representation chosen.

Figures 2, 3, and 4 list a program that makes use of the interface library routines to generate the SDDF file shown in figure 1. Error checking has been omitted from the program in the interest of brevity.

Referring to the code listing, lines 1–5 include necessary class definitions. The file *InitializeStatic.C*, included on line 6, defines and initializes the static data members used by classes in the SDDF library. The Stream Attribute is written by lines 15–17.

Lines 19–41 build up the structure for the **message send** Record Descriptor and write it to the SDDF file. Lines 42–45 create the data structures related to the **message send** event type that will be used to populate Record Data instances of that type. Program lines 46–63 build up the structure for the **context switch** Record Descriptor, write it to the output file, and create related data structures. Lines 65–70 and 88–91 populate and write the two **context switch** Record Data instances, while lines 71–80 and 81–87 produce the **message send** Record Data instances. Notice how field names are used when setting the field data values; for example, *timestamp* in the call on line 65 of the code.

To write the SDDF file in binary representation rather than ASCII, *BinaryPipeWriter.h* would be included instead of *AsciiPipeWriter.h*, and the class name *AsciiPipeWriter* on line 14 would be replaced with *BinaryPipeWriter*.

More details on the classes of the SDDF interface library are provided in Chapter 4. The library interface allows one to take advantage of the flexibility provided by the SDDF format without having to interpret the bytes of SDDF data directly.

```
18      /** Message Send Structure **/
19      attributes.clearEntries();
20      attributes.insert( "event", "message sent to one or more processors" );
21      structureP = new StructureDescriptor( "message send", attributes );

22          attributes.clearEntries();
23          fieldP = new FieldDescriptor( "timestamp", attributes, DOUBLE, 0 );
24          structureP->insert( *fieldP );
25          delete fieldP;

26          attributes.clearEntries();
27          attributes.insert( "Source", "Processor Sending Message" );
28          fieldP = new FieldDescriptor( "sourcePE", attributes, INTEGER, 0 );
29          structureP->insert( *fieldP );
30          delete fieldP;

31          attributes.clearEntries();
32          attributes.insert( "Destination", "Processor(s) Receiving Message" );
33          fieldP = new FieldDescriptor( "destinationPE", attributes, INTEGER, 1 );
34          structureP->insert( *fieldP );
35          delete fieldP;

36          attributes.clearEntries();
37          attributes.insert( "Size", "Message length in bytes" );
38          fieldP = new FieldDescriptor( "messageLength", attributes, INTEGER, 0 );
39          structureP->insert( *fieldP );
40          delete fieldP;

41      pipeWriter->putDescriptor( *structureP, 1 );
42      RecordDossier *messageRecord = new RecordDossier( 1, *structureP );
43      delete structureP;
44      Array *destinationPE = messageRecord->getArrayP( "destinationPE" );
45      int sz_destinationPE;

46      /** Context Switch Structure **/
47      attributes.clearEntries();
48      structureP = new StructureDescriptor( "context switch", attributes );

49          fieldP = new FieldDescriptor( "timestamp", attributes, DOUBLE, 0 );
50          structureP->insert( *fieldP );
51          delete fieldP;

52          attributes.clearEntries();
53          fieldP = new FieldDescriptor( "processor", attributes, INTEGER, 0 );
54          structureP->insert( *fieldP );
55          delete fieldP;
```

Figure 3: Program to generate sample SDDF file—part 2 of 3 parts.

```
56          fieldP = new FieldDescriptor( "processName", attributes, CHARACTER, 1 );
57          structureP->insert( *fieldP );
58          delete fieldP;

59      pipeWriter->putDescriptor( *structureP, 2 );
60      RecordDossier *contextSwitchRecord = new RecordDossier( 2, *structureP );
61      delete structureP;
62      Array *processName = contextSwitchRecord->getArrayP( "processName" );
63      int sz_processName;

64      /** Record Data Instances **/
65      contextSwitchRecord->setValue( "timestamp", 100.15 );
66      contextSwitchRecord->setValue( "processor", 2 );
67      sz_processName = 30;
68      processName->setDimSizes( &sz_processName );
69      processName->setCellString( "file i/o" );
70      pipeWriter->putData( *contextSwitchRecord );

71      messageRecord->setValue( "timestamp", 101.1 );
72      messageRecord->setValue( "sourcePE", 0 );
73      sz_destinationPE = 4;
74      destinationPE->setDimSizes( &sz_destinationPE );
75      destinationPE->setCellValue( 1, 0 );
76      destinationPE->setCellValue( 3, 1 );
77      destinationPE->setCellValue( 5, 2 );
78      destinationPE->setCellValue( 7, 3 );
79      messageRecord->setValue( "messageLength", 512 );
80      pipeWriter->putData( *messageRecord );


81      messageRecord->setValue( "timestamp", 102.15 );
82      messageRecord->setValue( "sourcePE", 7 );
83      sz_destinationPE = 1;
84      destinationPE->setDimSizes( &sz_destinationPE );
85      destinationPE->setCellValue( 1, 0 );
86      messageRecord->setValue( "messageLength", 1012 );
87      pipeWriter->putData( *messageRecord );

88      contextSwitchRecord->setValue( "timestamp", 108 );
89      contextSwitchRecord->setValue( "processor", 4 );
90      processName->setCellString( "idle" );
91      pipeWriter->putData( *contextSwitchRecord );

92      /** Close file **/
93      delete outFile;
94  }
```

Figure 4: Program to generate sample SDDF file—part 3 of 3 parts.

3 The Components of an SDDF File

This chapter describes the components of an SDDF file and presents in detail the meta-format syntax rules. The ideas introduced briefly in Chapter 2 are expanded, demonstrating the variety of information that can be represented and providing the knowledge necessary to create and interpret SDDF files. While the primary focus is on the ASCII representation, our discussion of the components and the information conveyed in individual SDDF records applies to the binary representation as well—exceptions to this will be noted. The syntax information presented in this chapter is stated concisely in the ASCII SDDF grammar found in Appendix A. The binary SSDF grammar is given in Appendix B.

3.1 SDDF File Headers

Each SDDF file begins with a header identifying it as either an ASCII or binary SDDF file. For an ASCII representation SDDF file, the header consists of the character string 'SDDFA'. The header for a binary representation SDDF file begins with the character string 'SDDFB'. The binary header also contains information identifying the byte order used in the file and other number representation specifications such as the size of the basic data types. The SDDF interface library routines use the byte order flag in the binary header to determine if byte swapping must be performed as the SDDF file is processed. [4] Appendix B gives the complete format of the binary SDDF file header.

3.2 SDDF Packet Types

After the SDDF header has been read, data in an SDDF file is processed one record at a time; a record is usually referred to as a packet. Thus, an SDDF file can be thought of as a stream of SDDF packets that are ordered based on their location in the file. The SDDF meta-format defines four distinct packet types: *Command*, *Stream Attribute*, *Record Descriptor*, and *Record Data*. In addition to prescribing the syntax for these packet types, the meta-format imposes a limited number of restrictions on the ordering and content of SDDF packets appearing in a single file.

To reiterate, the four packet types are the basic building blocks or components of an SDDF file. Instances of these components, what we commonly call records or packets, are grouped together in a specific order to produce an SDDF file. The SDDF syntax rules govern the form of the individual components and dictate valid relationships between component instances in a given SDDF file.

A packet usually consists of two parts: *packet header* and *packet body*. The packet header specifies the length, type, and tag of the packet. There is no tag for Stream Attribute packets. The packet body is interpreted differently for each packet type. The body may define a record descriptor, contain record data, or provide context information. There is no body associated with Command packets.

When using the binary SDDF format, the packet header appears explicitly at the beginning of each packet for performance reasons. With the ASCII format, the header can be reconstructed from the contents of the packet, relying on information from previously processed Record Descriptor packets to deduce the tag for Record Data packets. These encoding differences can be somewhat confusing if one attempts to manipulate SDDF packets directly. One advantage of using the SDDF

[4] Other than the byte order flag, the number representation specifications contained in the header are currently used only for error checking. Future releases may take advantage of these flags to perform more extensive automatic conversions on binary SDDF files.

library is that it provides a uniform interface to both ASCII and binary SDDF files. The library routines cache critical information from Record Descriptor packets as they are processed and use this information to interpret Record Data packets in the data stream, removing this burden from the programmer. In general, it is sufficient to understand what is meant by the packet type, tag, and body without being overly concerned about how this information is conveyed in the two representations. In this chapter, the ASCII representation is used to demonstrate the various packet types. The end of an ASCII packet is marked by two semicolon characters.

3.3 Command Packets

Command packets encapsulate commands. Each command is represented by a different tag. The Command packet has a null body, consisting only of a packet header. Command packets are intended to provide a mechanism for communicating between distinct processes sharing an SDDF data stream. For example, a command might be inserted into the SDDF file by a user program as it shifts from one phase of an application to another. This command could be recognized by a cooperating analysis module, causing it to flush running totals for the phase of the application just completed. Notice that interpretation of the tag's meaning is left up to the creator and the consumer of the Command packet; SDDF attaches no significance to the integer value contained in the packet.

```
%17: ;;
```

Figure 5: Command Packet—ASCII representation.

Figure 5 shows a Command packet with tag *17*. The '%' character flags this as a command packet, and the colon following the tag is required. Note that there is no packet body.

3.4 Stream Attribute Packets

Stream Attribute packets contain information pertinent to the entire trace file. Examples might include the date the file was created, the machine where the trace was generated, and the name of the program that generated the trace. Stream Attribute packets may be interspersed with other types of packets to provide "placeholders" in the file. For instance, a Stream Attribute packet could be used to mark the end of the records in an SDDF file, perhaps with the time the last record was written.

A Stream Attribute packet begins with the characters '/*' and ends with the characters '*/'. The information in a Stream Attribute packet is organized as one or more key/value attribute pairs, where both the key and the value are character strings enclosed in double quotes. In the ASCII version of SDDF, each attribute pair appears on a separate line in the record. There is no way to specify a tag in the ASCII representation for Stream Attribute packets. The binary representation does include the tag field as it has a fixed packet header format, but the value in the field is ignored by the SDDF interface library routines. For this reason, we say the tag is not used in Stream Attribute packets.

The Stream Attribute packet shown in Figure 6 contains two attribute pairs. The first has the key *creation date* and the value *March 1, 1992*, and the second has the key *machine* and the value *Intel iPSC/860*. SDDF library access methods support key-based searches to retrieve the value

```
/*
 * "creation date" "March 1, 1992"
 * "machine" "Intel iPSC/860"
 */ ;;
```

Figure 6: Stream Attribute Packet—ASCII representation.

of an attribute. An arbitrary number of attribute pairs can appear in a single Stream Attribute packet; there are no predefined keys or values.

3.5 Record Descriptor Packets

Record Descriptor packets describe record layouts, similar to structure declarations in the C programming language. Each Record Descriptor packet associates a particular tag and record name with a description of the fields that will appear in all data record instances having this tag. The tags and record names of all Record Descriptor packets must be unique in a given SDDF files, and the tags must be integers greater than zero.

The ASCII version of a Record Descriptor packet begins with the character '#' immediately followed by the tag and a colon character.

An optional *Record Attribute* may be associated with a Record Descriptor packet; if one exists it will appear after the tag. The Record Attribute contains information relevant to the entire record and is composed of one or more attribute key/value pairs. In the ASCII representation, each attribute pair appears on a separate line in the record and begins with two slash characters. As with Stream Attributes, the key and value are both character strings enclosed in double quotes.

The record name follows the tag and optional Record Attribute, and is given by a character string enclosed in double quotes. All Record Descriptor packets must have unique record names.

The final part of the Record Descriptor packet consists of one or more *Field Descriptors* enclosed in braces. [5]

Each Field Descriptor is composed of an optional Field Attribute, a required field type specifier, a required field name, an optional field dimension, and a terminating semicolon.

A *Field Attribute* may be associated with a given Field Descriptor, providing descriptive information about the field. If a Field Attribute exists, it appears at the beginning of the Field Descriptor and is composed of one or more attribute key/value pairs. As with the Record Attribute, each attribute pair appears on a separate line in the ASCII file and begins with two slash characters. The key and value are character strings enclosed in double quotes.

The field type specifier is required. Currently supported type specifiers are *char*, *int*, *float*, and *double* corresponding to the data types character, integer, float, and double in C or C++ . [6]

Following the field type specifier, the field name is given as a character string enclosed in double quotes. All Field Descriptors in a given Record Descriptor must have unique field names.

The last part of a Field Descriptor is the optional field dimension. If the field is a scalar quantity, the field dimension is not specified explicitly. If the field is an array, the number of dimensions

[5] It is actually possible to have a valid Record Descriptor packet with no Field Descriptors. The corresponding Record Data packets would contain no field values.

[6] We are planning add support for *short*, *long*, and *long double* types and the *unsigned* specifier in a future release.

in the array is indicated by an open/close square bracket pair for each dimension. Although the number of dimensions is specified for each field in the Record Descriptor, the number of elements in each dimension of an array quantity is not fixed at this point. Instead, Record Data packets corresponding to the structure defined by this Record Descriptor will contain not only values to populate the array cells, but also the size of each array dimension.

The declaration of dimension sizes in the data records allows array lengths to vary between data packets having a common structure definition and makes it unnecessary to preallocate arrays. As an example, consider a **`loop entry`** event with a vector data structure representing the number of times different loops have been entered. Each loop in the program being traced is assigned an index value. The vector cell value at a given offset corresponds to the number of times a particular loop has been entered. If we had to specify the size of the vector in the Record Descriptor, we would need to know the number of different loops that would be encountered at the time we declared the record structure for the **`loop entry`** event. If we guessed too small, we would not be able to save all the data. If we guessed too large, we would be wasting space reporting meaningless values. By specifying the vector length in the data record, we circumvent these problems and the vector is allowed to grow as appropriate for the data being reported. [7] There is no need to determine an appropriate size for the vector when the record structure is described, making structures more readily reusable; the same record structure and vector field is sufficient to report loop entry behavior for programs containing vastly different numbers of loops.

```
#1:
// "event" "message sent to one or more processors"
"message send" {
        // "From" "Processor Sending Message"
        int     "fromPE";
        // "To" "Processor(s) Receiving Message"
        int     "toPE"[];
        // "Size" "Message length in bytes"
        int     "messageLength";
        // "Type" "Character denoting message type"
        char    "messageType";
        // "Time" "Timestamp"
        // "Units" "msec"
        double  "time stamp";
};;
```

Figure 7: Record Descriptor Packet for **`message send`** event—ASCII representation.

Figure 7 shows a Record Descriptor packet with a tag of *1* and the record name **`message send`**. This packet has an associated Record Attribute with a single key/value pair. The attribute key is *event* and the attribute value is *message sent to one or more processors.* This Record Descriptor packet contains five Field Descriptors; each Field Descriptor is terminated by a semicolon character. All of the Field Descriptors in this packet have a Field Attribute providing information about the field. The first four Field Attributes have a single attribute key/value pair, and the last has two

[7] Note that SDDF also allows the size of the vector to decrease between successive data records in the data stream. That is unlikely to happen in the example presented.

attribute key/value pairs. The first and third Field Descriptors in the packet describe scalar values of type integer with names *fromPE* and *messageLength*. The second Field Descriptor details an integer data type whose name is *toPE*. It is an array value of dimension one (a vector of integers). The fourth field is named *messageType* and will contain a single character. Finally, the field *time stamp* is a scalar of type double.

```
#9:
"procedure call summary" {
        char    "data file"[];
        char    "procedures"[][];
        // "dimension 0" "procedure"
        // "dimension 1" "processor"
        // "cell value" "count"
        int     "count of calls"[][];
};;
```

Figure 8: Record Descriptor Packet for **procedure call summary** event—ASCII representation.

Figure 8 shows a second Record Descriptor packet with no associated Record Attribute. The tag is *9*, the record name is **procedure call summary**, and there are three Field Descriptors. The first field describes a one dimension array of characters (i.e., a character string) with the name *data file*. The second field specifies a two dimensional array of characters (a vector of character strings) named *procedures*. The last field has a Field Attribute containing three key/value attribute pairs. The field type specifier is *int*, the field name is *count of calls*, and the dimension is two.

3.6 Record Data Packets

The final SDDF packet type is the *Record Data* packet. Packets of this type contain the actual data to be processed, as opposed to commands, comments, or specifications of how data should be formatted. In the ASCII version, each Record Data packet has a record name that must match the record name of a Record Descriptor packet that preceded it in the SDDF file. In the binary SDDF version, packet tags are matched rather than record names. The body of the Record Data packet contains values for each field described in the corresponding Record Descriptor. The packet body is enclosed in braces, and field values are separated by commas.

For non-scalar fields, the field value is composed of several parts. First, for each dimension of the array, the number of elements in that dimension is given. Following that, braces surround the cell data values, listed in row-major order and separated by commas. Character strings are enclosed in double quotes.

```
"message send" { 0,
        [4] { 1, 3, 5, 7 },
        512, B, 100.100000 };;
```

Figure 9: Record Data Packet for **message send** event—ASCII representation.

Figure 9 shows a Record Data packet with the record name **message send**. The name matches

that of the Record Descriptor packet in Figure 7; the body of this Record Data packet is interpreted based on the record layout described in that Record Descriptor. The value of the integer field *fromPE* is *0*. The *toPE* vector has four elements with integer values *1*, *3*, *5*, and *7*. The *messageLength* field has the integer value *512*. The *messageType* field has the character value *B*, and the double precision floating point value *100.100000* is assigned to the field *time stamp*.

```
"procedure call summary" {
        [30] { "/mnt/data/testdata-1" },
        [3][20] { "compare()", "swap()", "error()" },
        [3][4] { 15, 10, 2, 20, 10, 7, 2, 5, 8, 3, 17, 2 } };;
```

Figure 10: Record Data Packet for **procedure call summary** event—ASCII representation.

Figure 10 shows a Record Data packet corresponding to the Record Descriptor packet in Figure 8. Notice for all character string fields, the number of characters enclosed in double quotes is less than the number of elements allocated to hold the string: */mnt/data/testdata-1* is shorter than 30 characters; *compare()*, *swap()*, and *error()* are each less than 20 characters. When the length of the field value is less that the declared length of the character string field, the internal representation of the field is padded to the specified length (30 and 20 in this case). It is an error if the string contains more characters than the element count.

Using the descriptive information contained in the Field Attribute for the *count of calls* field, together with the values in this Record Data packet, one could infer that the procedure *swap* was called seven times on the second processor when the program producing the trace was executed using the data file */mnt/data/testdata-1*. Of course, the validity of that inference depends on whether the correct semantic meaning has been attached to the data—that is outside the scope of the SDDF specification.

4 The SDDF Interface Library

The SDDF interface library consists of several C++ classes that provide a framework for interacting with ASCII and binary SDDF files. It is not necessary to fully understand all of the classes in the library to work comfortably with SDDF files.

A brief introduction to the library routines was presented in Section 2.2—*A Sample Program using the SSDF Interface Library.* This chapter provides greater detail, using textual explanations together with code fragments to illustrate the classes and methods of importance to someone creating SDDF files. Many of the code fragments included in this chapter are taken from programs listed in their entirety in Appendices C and D; in those cases, the line numbers shown with the code fragments correspond to the line numbers appearing in the Appendicies.

Classes used to interact with SDDF files are presented first, followed by those used to save and manipulate the information conveyed by the individual SDDF packets. Complete documentation of the classes and methods can be found in the accompanying document [4]. [8]

4.1 InputFileStreamPipe and OutputFileStreamPipe

The *InputFileStreamPipe* and *OutputFileStreamPipe* classes are used to open files for input and output. They provide interfaces to standard Unix files that are compatible with the SDDF PipeReader and PipeWriter classes we will see later.

The relevant methods are the *constructor* which opens the file, the *successfulOpen* method which checks to see if the open completed without error, and the *destructor* which closes the file. Use InputFileStreamPipe when reading files (open for input), and OutputFileStreamPipe when writing files (open for output). File access through these classes is buffered to improve performance. The default buffer size of 204,800 bytes may be overridden by passing the desired buffer size as the second argument to the FileStreamPipe constructor. The FileStreamPipe methods are demonstrated in the code presented in Figure 11.

The method *createPipeReader* is also important; it is explained in the PipeReader section which follows.

4.2 PipeReader

The *PipeReader* class is used to read SDDF data packets incrementally for processing. For each SDDF packet, an initial call is made to retrieve the packet header. Based on the packet type contained in the packet header, a second call is made to load the packet body into an appropriate data structure. If the packet is a Command Packet, the second call is not necessary because there is no packet body. The derived *AsciiPipeReader*, *BinaryPipeReader* and *ConversionPipeReader* classes support ASCII, binary, and binary byte-swapped SDDF representations. While these derived classes are important, they are rarely accessed directly by a user of the SDDF interface library.

To construct a PipeReader for a given file, a call is made to the *createPipeReader* method of the InputFileStreamPipe instance associated with the file that is to be read. This method uses the header information in the file to determine its SDDF representation and creates a PipeReader to match. If for some reason the PipeReader cannot be created, an error is displayed and NULL is returned by the *createPipeReader* method.

[8] Those interested in the nitty-gritty details (that object-oriented programming is supposed to hide) are encouraged to read the header files and source code for the classes in the library.

```
11 main()
12 {
13    /** Open input and output files **/
14    InputFileStreamPipe *inFile = new InputFileStreamPipe( "/tmp/file.input" );
15    if ( ! inFile->successfulOpen() ) {
16       cerr << "ERROR opening input file\n";
17       exit( -1 );
18    }
19    OutputFileStreamPipe *outFile = new OutputFileStreamPipe( "/tmp/file.binary" );
20    if ( ! outFile->successfulOpen() ) {
21       cerr << "ERROR opening output file\n";
22       delete inFile;
23       exit( -2 );
24    }
            ⋮

67    /** Close files and exit **/
68    delete outFile;
69    delete inFile;
70    exit( 0 );
71 }
```

Figure 11: Code fragment (Appendix C) showing FileStreamPipe classes.

The *getPacketHeader* method of the PipeReader class reads the next packet header from the file—when the end of file is reached, a dummy packet header with the type `PIPE_EMPTY` is returned.

If the packet is a Command packet, there is no packet body. If it is a Stream Attribute, Record Descriptor, or Record Data packet, the corresponding PipeReader method *getAttributes*, *getDescriptor*, or *getData* is called to load the packet body into an appropriate data structure where the information that was encapsulated in the packet can be accessed and manipulated.

4.3 AsciiPipeWriter, BinaryPipeWriter, and ConversionPipeWriter

The classes used to assemble and write SDDF packets are *AsciiPipeWriter*, *BinaryPipeWriter*, and *ConversionPipeWriter*. The usage and methods of the PipeWriter classes are identical—the three classes support the ASCII, binary, and binary byte-swapped SDDF representations. The PipeWriter *constructor* takes a pointer to an OutputFileStreamPipe instance and writes data to the file associated with that OutputFileStreamPipe. The four methods *putCommand*, *putAttributes*, *putDescriptor*, and *putData* are used to write Command, Stream Attribute, Record Descriptor, and Record Data packets. The *putCommand* method takes the packet tag as an argument; the other methods each take a pointer to an object instance containing the information that is to be encapsulated in the SDDF packet.

The code fragment shown in Figure 12 demonstrates the PipeReader and BinaryPipeWriter classes and methods being used to translate an SDDF file of any type (ASCII, binary, or binary byte-swapped) into an equivalent binary SDDF file. The methods of the PipeReader and PipeWriter classes hide the details of processing ASCII, binary, or binary byte-swapped SDDF packets, as well

```
25    /** Create PipeReader and PipeWriter instances with types matching the    **
26     ** SDDF representation of the input and output files.                    **/
27    PipeReader *reader = inFile->createPipeReader();
28    if ( reader == NULL ) {
29       cerr << "ERROR connecting PipeReader to input file\n";
30       delete outFile;
31       delete inFile;
32       exit( -3 );
33    }
34    PipeWriter *writer = new BinaryPipeWriter( outFile );

35    /** RecordDictionary instance holds persistent data.                      **
36     ** Read from the input file until it is empty.  Process each input       **
37     ** packet based on the type.  Write corresponding packet to output.      **/
38    RecordDictionary RD;
39    Attributes *Ap;
40    StructureDescriptor *SDp;

41    PacketHeader PH = reader->getPacketHeader();

42    while( PH.type != PIPE_EMPTY ) {
43       switch( PH.type ) {
44         case PKT_COMMAND:
45            writer->putCommand( PH.tag );
46            break;
47         case PKT_ATTRIBUTE:
48            Ap = new Attributes();
49            reader->getAttributes( *Ap );
50            writer->putAttributes( *Ap );
51            delete Ap;
52            break;
53         case PKT_DESCRIPTOR:
54            SDp = new StructureDescriptor;
55            reader->getDescriptor( *SDp );
56            writer->putDescriptor( *SDp, PH.tag );
57            RD.insert( PH.tag, *SDp );
58            delete SDp;
59            break;
60         case PKT_DATA:
61            reader->getData( RD.fetch( PH.tag ) );
62            writer->putData( RD.fetch( PH.tag ) );
63            break;
64       }
65       PH = reader->getPacketHeader();
66    }
```

Figure 12: Code fragment (Appendix C) showing PipeReader and PipeWriter classes.

```
55      Attributes              attributes;
             ⋮
60      /** Stream Attribute **/
61      attributes.insert( "Creation Date", "Feb. 1, 1992" );
62      attributes.insert( "Machine", "Intel iPSC/860" );
63      pipeWriter->putAttributes( attributes );
```

Figure 13: Code fragment (Appendix D) showing Attributes class.

as the details of packet decomposition and recomposition. The Attributes, RecordDictionary, and StructureDictionary classes will be covered in more detail later in the chapter—they are used to store and manipulate the information communicated by SDDF packets.

4.4 CString

The *CString* class is an abstract data type for storing and manipulating character strings. The class provides comparison operators (`<`, `<=`, `>`, `>=`, `!=`, `==`), the assignment operator (`=`), and defines conversion operators between the CString and `char*` data types. CString objects are often data members or method arguments for more complex classes in the library. In most of these cases, a `char*` type can be used and the conversion routine will automatically generate the appropriate CString instance.

4.5 Attributes

The Stream Attribute, Record Attribute, and Field Attribute described in Chapter 3 can all be represented internally as instances of the *Attributes* class. The *insert* member function is used to add attribute key/value pairs to the instance; the *clearEntries* method clears all key/value pairs in the instance.

The code segment in Figure 13 creates an Attributes instance with the name `attributes` and populates it with two key/value pairs. The PipeWriter method *putAttributes* writes the contents of `attributes` as an SDDF Stream Attribute packet.

4.6 FieldDescriptor

The *FieldDescriptor* class is used to define a field by specifying its name, attributes, type, and dimension. The *constructor* used most frequently takes four arguments: the field name; a reference to an Attributes instance containing the Field Attribute; the field type (`CHARACTER`, `INTEGER`, `FLOAT`, or `DOUBLE`); and the number of dimensions—zero if the value is a scalar.

Figure 14 shows the constructors for two fields. The first field, *sourcePE*, is a scalar integer with an associated Field Attribute (code lines 68-70). The second field, *destinationPE*, is a one-dimensional array of type integer that also has an associated Field Attribute (code lines 73-75).

4.7 StructureDescriptor

The *StructureDescriptor* class is used to build up data record structures, providing an internal representation for the information conveyed by an SDDF Record Descriptor packet. The *constructor*

```
56      StructureDescriptor     *structureP;
57      FieldDescriptor         *fieldP;
58      int                     tag = 1;
              ⋮
64      /* Message Send structure */
65      attributes.clearEntries();
66      attributes.insert( "Event", "Message sent to one or more processors" );
67      structureP = new StructureDescriptor( "message send", attributes );

68          attributes.clearEntries();
69          attributes.insert( "Source", "Processor Sending Message" );
70          fieldP = new FieldDescriptor( "sourcePE", attributes, INTEGER, 0 );
71          structureP->insert( *fieldP );
72          delete fieldP;

73          attributes.clearEntries();
74          attributes.insert( "Destination", "Processor(s) Receiving Message" );
75          fieldP = new FieldDescriptor( "destinationPE", attributes, INTEGER, 1 );
76          structureP->insert( *fieldP );
77          delete fieldP;
              ⋮
94      pipeWriter->putDescriptor( *structureP, tag );
```

Figure 14: Code fragment (Appendix D) showing FieldDescriptor and StructureDescriptor classes.

arguments are the record name and a reference to an Attributes instance containing the Record Attribute. The method *insert* adds new FieldDescriptors into the record structure. Fields occur in the data record structure in the same order they are inserted.

Code line 67 in Figure 14 shows the creation of a StructureDescriptor instance named **message send** that has an associated Record Attribute with one key/value pair. Fields are added to the data record structure on lines 71 and 76. The PipeWriter method *putDescriptor* is called on line 94 to write the contents of the StructureDescriptor instance as a Record Descriptor packet with tag *1* and record name **message send**.

4.8 Value

The *Value* class defines a common interface to all data types supported by the SDDF specification. The class provides an internal representation for data values corresponding to fields of types **CHARACTER**, **INTEGER**, **FLOAT**, and **DOUBLE**. The member functions support manipulation of the values, conversion between the C++ types *char*, *int*, *float*, and *double* and the internal Value representations, and comparison operations. In general, the C++ types can be used whenever methods specify arguments of type Value—the conversion routines will automatically generate the necessary Value instances.

4.9 Array

A field with dimension greater than zero represents an array Value and is supported by the SDDF library class *Array*. If a Value instance represents a non-scalar amount, it's internal representation includes a pointer to the Array instance holding the data values for the array elements.

For the purpose of creating SDDF files, the relevant Array methods are the *constructor*, *setDimSizes*, *setCellValue*, and *setCellString*. The *constructor* takes the data type (`CHARACTER`, `INTEGER`, `FLOAT`, or `DOUBLE`) and the array dimension as arguments. Before an Array instance can be populated, it is necessary to specify the number of cells in each dimension using the *setDimSizes* method. *setDimSizes* takes a single argument—a C++ array of integers having at least as many elements as the Array instance has dimensions. Each element of the C++ integer array specifies the number of cells in the corresponding dimension of the Array instance. The size of the dimensions in an Array can be changed over the lifetime of the instance by making additional calls to the *setDimSizes* method.

The member function *setCellValue* is used to set the value of a given cell in the Array instance. The first argument is the Value the cell should assume; subsequent arguments give the indices into the Array instance. The number of indices should equal the number of dimensions in the instance. A given index should not exceed the valid range for the dimension it is specifying as set in the most recent call to *setDimSizes*. Figure 15 shows an example of the *setDimSizes* and *setCellValue* methods.

```
1    int   dimSizes[3] = { 2, 1, 3 };

2    Array *threeD = new Array( INTEGER, 3 );
3    threeD->setDimSizes( dimSizes );
4    threeD->setCellValue( 10, 0, 0, 0 );
5    threeD->setCellValue( 20, 0, 0, 1 );
6    threeD->setCellValue( 30, 0, 0, 2 );
7    threeD->setCellValue( 40, 1, 0, 0 );
8    threeD->setCellValue( 50, 1, 0, 1 );
9    threeD->setCellValue( 60, 1, 0, 2 );
```

Figure 15: Code fragment showing Array methods *setDimSizes* and *setCellValue*.

The method *setCellString* provides a mechanism for setting the elements in Array instances representing character strings. Rather than make individual calls to *setCellValue* for each character in a string, a single call per string is made to *setCellString*. The first argument is either a CString reference or a pointer to a character string. Subsequent arguments give the indices into the Array instance where the string should be stored. If the dimension of the Array is `N`, there will be `N-1` indices in the *setCellString* call. If the length of the new character string is less than the size of the highest Array dimension, the internal representation of the string is padded with null characters.

Figure 16 shows two Array instances of type `CHARACTER`. The first holds a single character string of up to 30 characters. The second holds three strings, each having up to 20 characters.

```
 1      int strLength = 30;
 2      int dimSizes[2] = { 3, 20 };

 3      Array *oneString = new Array( CHARACTER, 1 );
 4      oneString->setDimSizes( &strLength );
 5      oneString->setCellString( "Here is my character string." );

 6      Array *moreStrings = new Array( CHARACTER, 2 );
 7      moreStrings->setDimSizes( dimSizes );
 8      moreStrings->setCellString( "First string here.", 0 );
 9      moreStrings->setCellString( "Second string here.", 1 );
10      moreStrings->setCellString( "Third string here.", 2 );
```

Figure 16: Code fragment showing Array method *setCellString*.

4.10 RecordDossier

Once a data record structure has been constructed in a StructureDescriptor instance, the *RecordDossier* class provides a mechanism for caching the structure information and managing data values for the fields in the record. The RecordDossier methods use cached field names, types, and dimensions to manipulate the field values.

The RecordDossier *constructor* takes two arguments—the tag associated with the data record structure (the same tag used earlier in the PipeWriter method *putDescriptor*), and the StructureDescriptor instance containing the record structure. After a RecordDossier instance has been created, the *setValue* member function is used to set scalar field values. For non-scalar fields (those whose dimension is greater than zero), the RecordDossier method *getArrayP* returns a pointer to the Array instance representing the field. The Array methods *setDimSizes*, *setCellValue*, and *setCellString* can then be used to set the dimension sizes and cell values for the field.

Line 95 of the code fragment shown in Figure 17 demonstrates the creation of a new RecordDossier instance to store the **message send** record structure built up previously in Figure 14. The method *getArrayP* called on line 96 retrieves an Array pointer for the non-scalar field *destinationPE*. The scalar fields are populated with values on lines 156, 163, 164, and 165. Lines 157-162 set the number of elements and individual cell values for the *destinationPE* field using the Array pointer. The PipeWriter method *putData* is called on line 166 to write the corresponding SDDF Record Data packet.

4.11 RecordDictionary

When there are many different data record structures being used, it is cumbersome to maintain RecordDossier instances for each record type. The *RecordDictionary* class is useful in these situations.

The RecordDictionary class maintains a set of RecordDossiers which can be accessed by their record tags. The *insert* member function takes a tag and a StructureDescriptor instance reference—exactly like the RecordDossier *constructor* method. A call to *insert* creates a RecordDossier instance inside the RecordDictionary—the RecordDossier instance can then be retrieved from the RecordDictionary with the *fetch* method which takes the tag as its argument. The RecordDictionary class

```
95      RecordDossier *messageRecord = new RecordDossier( tag, *structureP );
96      Array *destinationPE = messageRecord->getArrayP( "destinationPE" );
              ⋮
155     /** Message Send Data Packet **/
156     messageRecord->setValue( "sourcePE", 0 );
157     dimSizes[0] = 4;
158     destinationPE->setDimSizes( dimSizes );
159     destinationPE->setCellValue( 1, 0 );
160     destinationPE->setCellValue( 3, 1 );
161     destinationPE->setCellValue( 5, 2 );
162     destinationPE->setCellValue( 7, 3 );
163     messageRecord->setValue( "messageLength", 512 );
164     messageRecord->setValue( "messageType", 'B' );
165     messageRecord->setValue( "timestamp", 100.1 );
166     pipeWriter->putData( *messageRecord );
```

Figure 17: Code fragment (Appendix D) showing RecordDossier class.

also has the member functions *setValue* and *getArrayP*, analogous to the methods in the RecordDossier class. The RecordDictionary *setValue* and *getArrayP* methods take the record tag as the first argument—subsequent arguments are the same as the RecordDossier methods.

Figure 12, shown earlier in the chapter, demonstrates the RecordDictionary class being used to simplify data management when translating an ASCII SDDF file to a binary SDDF file. The RecordDictionary instance is created on line 38. When a RecordDescriptor packet is seen, the record structure is cached in the RecordDictionary on line 57 of the code. Referring to lines 61 and 62 for `PKT_DATA` packets, the RecordDictionary *fetch* method is called with the tag from the packet header to access the corresponding RecordDossier instance in the RecordDictionary. The PipeReader *getData* method updates the field values in the RecordDossier instance with the information from the packet body. On line 62, the PipeWriter *putData* method encapsulates the information from the RecordDossier instance into an SDDF packet and writes it to the output.

A Grammar for SDDF File - ASCII Representation

This appendix contains a grammar specifying the syntax of an ASCII representation SDDF file. A modified Backus-Naur notation is used. The symbol □ is used as a shorthand notation for optional white space. Characters which must appear verbatim are shown in the typewriter font. Comments are italicized.

A.1 Data File Production Rules

⟨data file⟩ —→ ⟨ascii header⟩ ⟨packet⟩*

⟨ascii header⟩ —→ `SDDFA`

⟨packet⟩ —→ ⟨stream attribute⟩ | ⟨record descriptor⟩ | ⟨record data⟩ | ⟨command⟩

A.2 Stream Attribute Production Rules

⟨stream attribute⟩ —→ □ `/*` ⟨one stream attribute⟩* □ `*/` □ `;;`

⟨one stream attribute⟩ —→ □ `*` □ ⟨string⟩ □ ⟨string⟩

A.3 Record Descriptor Production Rules

⟨record descriptor⟩ —→ □ `#` ⟨tag⟩ `:` ⟨attribute⟩* ⟨record name⟩ { ⟨field descriptor⟩* } □ `;;`

⟨attribute⟩ —→ □ `//` □ ⟨string⟩ □ ⟨string⟩

⟨field descriptor⟩ —→ ⟨attribute⟩* ⟨field type⟩ ⟨white space⟩$^+$ ⟨field name⟩ ⟨a dimension⟩* `;` □

⟨field type⟩ —→ □ ⟨valid type⟩

⟨valid type⟩ —→ `char` | `int` | `float` | `double`

⟨field name⟩ —→ □ ⟨string⟩

⟨a dimension⟩ —→ `[]`

A.4 Record Data Production Rules

⟨record data⟩ —→ ⟨record name⟩ { ⟨field value list⟩ } □ `;;`

⟨field value list⟩ —→ □ | □ ⟨field value⟩ ⟨another field value⟩* □

> *NOTE: The ⟨field value list⟩ should contain a ⟨field value⟩ for each ⟨field descriptor⟩ defined in the ⟨record descriptor⟩ having the same ⟨record name⟩.*

⟨another field value⟩ —→ `,` □ ⟨field value⟩

⟨field value⟩ —→ ⟨scalar value⟩ | ⟨array value⟩

⟨scalar value⟩ —→ ⟨string⟩ | ⟨character⟩ | ⟨integer⟩ | ⟨float⟩ | ⟨double⟩

⟨array value⟩ —→ ⟨dimension size⟩$^+$ □ ⟨array value list⟩

NOTE: The ⟨array value⟩ must have a ⟨dimension size⟩ for each ⟨a dimension⟩ defined in the ⟨field descriptor⟩ for this field in the ⟨record descriptor⟩ with the same ⟨record name⟩.

⟨dimension size⟩ —→ [⟨non-negative integer⟩]

⟨array value list⟩ —→ { □ ⟨scalar value⟩ ⟨another scalar value⟩* ⟨white space⟩$^+$ }

NOTE: The number of ⟨scalar value⟩s in the ⟨array value list⟩ must equal the product of all the ⟨dimension size⟩ entries for this ⟨array value⟩.

⟨another scalar value⟩ —→ , □ ⟨scalar value⟩

A.5 Command Production Rule

⟨command⟩ —→ □ % ⟨tag⟩ : □ ;;

A.6 Miscellaneous Production Rules

⟨record name⟩ —→ □ ⟨string⟩ □

⟨tag⟩ —→ □ ⟨positive integer⟩

□ —→ ⟨white space⟩*

⟨white space⟩ —→ SPACE | TAB | NEWLINE | FORMFEED | VERTICAL_TAB

⟨string⟩ —→ " ⟨string character⟩* "

⟨string character⟩ —→ *any ASCII character except a quote (") or a backslash (\)*
| ⟨escaped character⟩

⟨escaped character⟩ —→ \ ⟨character⟩

⟨character⟩ —→ *any ASCII character*

⟨positive integer⟩ —→ *an* ⟨integer⟩ *greater than zero*

⟨non-negative integer⟩ —→ *an* ⟨integer⟩ *greater than or equal to zero*

⟨integer⟩ —→ *a decimal number accepted by scanf(3s) format %d*

⟨float⟩ —→ *a floating point number accepted by scanf(3s) format %f*

⟨double⟩ —→ *a double-precision floating point number accepted by scanf(3s) format %lf*

B Grammar for SDDF File - Binary Representation

This appendix contains a grammar specifying the syntax of a binary representation SDDF file. A modified Backus-Naur notation is used. Comments appear italicized. Machine-dependent data types are indicated by their C++ type name and are set in the typewriter font: `char`, `int`, `float` and `double`. Constants appear as upper-case names in the typewriter font—the constant values are given in the last two sections of this appendix.

B.1 Data File Production Rule

⟨data file⟩ —→ ⟨binary header⟩ ⟨packet⟩*

B.2 Binary Header Production Rules

⟨binary header⟩ —→ ⟨sddf binary indicator⟩ ⟨bytes in header⟩ ⟨byte order⟩ ⟨integer format⟩
⟨sddf binary indicator⟩ ⟨float format⟩ ⟨double format⟩ ⟨long double format⟩
⟨character format⟩ ⟨sizeof char⟩ ⟨sizeof short⟩ ⟨sizeof int⟩ ⟨sizeof long⟩ ⟨sizeof float⟩
⟨sizeof double⟩ ⟨sizeof long double⟩

NOTE: ⟨long double format⟩, ⟨sizeof short⟩, ⟨sizeof long⟩, and ⟨sizeof long double⟩ are included for completeness but their values are not used in the current release.

⟨sddf binary indicator⟩ —→ SDDFB *(this represents a verbatim character string)*

⟨bytes in header⟩ —→ ⟨numeric-valued byte⟩

⟨byte order⟩ —→ `BIG_ENDIAN` | `LITTLE_ENDIAN`

⟨integer format⟩ —→ `TWOS_COMPLEMENT` | `ONES_COMPLEMENT` | `SIGN_MAGNITUDE`

⟨float format⟩ —→ `IEEE` | `VAX` | `IBM` | `CRAY`

⟨double format⟩ —→ `IEEE` | `VAX` | `IBM` | `CRAY`

⟨long double format⟩ —→ `IEEE` | `VAX` | `IBM` | `CRAY`

⟨character format⟩ —→ `ASCII` | `EBCDIC`

⟨sizeof char⟩ —→ ⟨numeric-valued byte⟩

⟨sizeof short⟩ —→ ⟨numeric-valued byte⟩

⟨sizeof int⟩ —→ ⟨numeric-valued byte⟩

⟨sizeof long⟩ —→ ⟨numeric-valued byte⟩

⟨sizeof float⟩ —→ ⟨numeric-valued byte⟩

⟨sizeof double⟩ —→ ⟨numeric-valued byte⟩

⟨sizeof long double⟩ —→ ⟨numeric-valued byte⟩

B.3 Packet Production Rules

⟨packet⟩ —→ ⟨packet length⟩ ⟨packet type_tag_data⟩

⟨packet length⟩ —→ ⟨positive integer⟩

> *NOTE: The ⟨packet length⟩ is the number of bytes in the ⟨packet⟩, including those used to hold the ⟨packet length⟩.*

⟨packet type_tag_data⟩ —→ ⟨stream attribute⟩ | ⟨record descriptor⟩ | ⟨record data⟩ | ⟨command⟩

B.4 Stream Attribute Production Rule

⟨stream attribute⟩ —→ `PKT_ATTRIBUTE` ⟨tag⟩ ⟨attribute list⟩

> *NOTE: The value of ⟨tag⟩ is never used in the ⟨stream attribute⟩ packet, but a "dummy tag" must be included to preserve proper alignment. In a ⟨stream attribute⟩ packet, the ⟨tag⟩ may be zero.*

B.5 Record Descriptor Production Rules

⟨record descriptor⟩ —→
`PKT_DESCRIPTOR` ⟨tag⟩ ⟨record name⟩ ⟨attribute list⟩ ⟨field count⟩ ⟨field descriptor⟩*

> *NOTE: The number of ⟨field descriptor⟩s must equal the ⟨field count⟩.*

⟨record name⟩ —→ ⟨string⟩

⟨field count⟩ —→ ⟨non-negative integer⟩

⟨field descriptor⟩ —→ ⟨field name⟩ ⟨attribute list⟩ ⟨field type⟩ ⟨field dimension⟩

⟨field name⟩ —→ ⟨string⟩

⟨field type⟩ —→ `CHARACTER` | `INTEGER` | `FLOAT` | `DOUBLE`

⟨field dimension⟩ —→ ⟨non-negative integer⟩

B.6 Record Data Production Rules

⟨record data⟩ —→ `PKT_DATA` ⟨tag⟩ ⟨record data values⟩

⟨record data values⟩ —→ *byte stream*

> *The record data values are represented as a byte stream containing a value for each field described in the ⟨record descriptor⟩ whose ⟨tag⟩ is the same as the ⟨tag⟩ of this ⟨record data⟩ packet. For each field, the ⟨field type⟩ is used to determine the number of bytes required to represent the field value. The field values are written as machine-dependent* `char`, `int`, `float`, *and* `double` *quantities. For non-scalar fields (those with a ⟨field dimension⟩ greater than zero), the field value consists of ⟨field dimension⟩* `int`s *giving the size for each dimension, followed by the actual cell values in row-major order—the number of bytes taken up by each cell value depends on the ⟨field type⟩.*

B.7 Command Production Rule

⟨command⟩ —→ `PKT_COMMAND` ⟨tag⟩

B.8 Attribute List Production Rules

⟨attribute list⟩ —→ ⟨attribute count⟩ ⟨attribute pair⟩*

NOTE: The number of ⟨attribute pair⟩s must equal the ⟨attribute count⟩.

⟨attribute count⟩ —→ ⟨non-negative integer⟩

⟨attribute pair⟩ —→ ⟨attribute name⟩ ⟨attribute value⟩

⟨attribute name⟩ —→ ⟨string⟩

⟨attribute value⟩ —→ ⟨string⟩

B.9 Miscellaneous Production Rules

⟨tag⟩ —→ ⟨positive integer⟩

⟨positive integer⟩ —→ `int` *with value greater than zero*

⟨non-negative integer⟩ —→ `int` *with value greater than or equal to zero*

⟨numeric-valued byte⟩ —→ *a single-byte quantity with a decimal value ranging from 0 to 127*

⟨string⟩ —→ *a null-terminated sequence of zero or more* `char`*s*

B.10 Integer Constants

Packet Types:

`PKT_ATTRIBUTE` = 1
`PKT_DESCRIPTOR` = 2
`PKT_DATA` = 3
`PKT_COMMAND` = 4

Data Types:

`CHARACTER` = 0
`INTEGER` = 1
`FLOAT` = 2
`DOUBLE` = 3

B.11 Numeric Constants Represented as Bytes

Byte Order:

`BIG_ENDIAN` = 0
`LITTLE_ENDIAN` = 1

Integer Format:

`TWOS_COMPLEMENT` = 0
`ONES_COMPLEMENT` = 1
`SIGN_MAGNITUDE` = 2

Floating Point Representation:

`IEEE` = 0
`VAX` = 1
`IBM` = 2
`CRAY` = 3

C Program to Convert SDDF File to Binary Representation

The code in this appendix can be linked with the Pablo SDDF interface library to produce an executable that will take an ASCII or binary (native or non-native byte order) representation SDDF file as input and produce an equivalent binary (native byte order) representation SDDF file. The input and output file names are hard-coded in the program (`/tmp/file.input` and `/tmp/file.binary`) to minimize the code devoted to non-SDDF specific processing. The program demonstrates the use of the *FileStreamPipe*, *PipeReader* and *PipeWriter* classes. The *Attributes*, *RecordDictionary*, and *StructureDescriptor* classes are also introduced—they are used more extensively in the program presented in Appendix D.

```
 1 #include <stdlib.h>
 2 #include "Attributes.h"
 3 #include "BinaryPipeWriter.h"
 4 #include "InputFileStreamPipe.h"
 5 #include "OutputFileStreamPipe.h"
 6 #include "PacketHeader.h"
 7 #include "PipeReader.h"
 8 #include "RecordDictionary.h"
 9 #include "StructureDescriptor.h"
10 #include "InitializeStatic.C"

11 main()
12 {
13    /** Open input and output files **/
14    InputFileStreamPipe *inFile = new InputFileStreamPipe( "/tmp/file.input" );
15    if ( ! inFile->successfulOpen() ) {
16       cerr << "ERROR opening input file\n";
17       exit( -1 );
18    }
19    OutputFileStreamPipe *outFile = new OutputFileStreamPipe( "/tmp/file.binary" );
20    if ( ! outFile->successfulOpen() ) {
21       cerr << "ERROR opening output file\n";
22       delete inFile;
23       exit( -2 );
24    }

25    /** Create PipeReader and PipeWriter instances with types matching the    **
26     ** SDDF representation of the input and output files.                    **/
27    PipeReader *reader = inFile->createPipeReader();
28    if ( reader == NULL ) {
29       cerr << "ERROR connecting PipeReader to input file\n";
30       delete outFile;
31       delete inFile;
32       exit( -3 );
33    }
34    PipeWriter *writer = new BinaryPipeWriter( outFile );
```

```
35    /** RecordDictionary instance holds persistent data.                    **
36     ** Read from the input file until it is empty.  Process each input     **
37     ** packet based on the type.  Write corresponding packet to output.    **/
38    RecordDictionary RD;
39    Attributes *Ap;
40    StructureDescriptor *SDp;

41    PacketHeader PH = reader->getPacketHeader();

42    while( PH.type != PIPE_EMPTY ) {
43       switch( PH.type ) {
44         case PKT_COMMAND:
45            writer->putCommand( PH.tag );
46            break;
47         case PKT_ATTRIBUTE:
48            Ap = new Attributes;
49            reader->getAttributes( *Ap );
50            writer->putAttributes( *Ap );
51            delete Ap;
52            break;
53         case PKT_DESCRIPTOR:
54            SDp = new StructureDescriptor;
55            reader->getDescriptor( *SDp );
56            writer->putDescriptor( *SDp, PH.tag );
57            RD.insert( PH.tag, *SDp );
58            delete SDp;
59            break;
60         case PKT_DATA:
61            reader->getData( RD.fetch( PH.tag ) );
62            writer->putData( RD.fetch( PH.tag ) );
63            break;
64       }
65       PH = reader->getPacketHeader();
66    }

67    /** Close files and exit **/
68    delete outFile;
69    delete inFile;
70    exit( 0 );
71 }
```

D Program to Generate a Sample SDDF File

The code in this appendix can be linked with the SDDF interface library to produce an executable that will create an SDDF representation file. The ASCII representation SDDF file shown in Appendix E was generated by this program. The program demonstrates the manipulation of all supported data types.

```
 1 #include <stream.h>

 2 #include "AsciiPipeWriter.h"
 3 #include "Attributes.h"
 4 #include "BinaryPipeWriter.h"
 5 #include "ConversionPipeWriter.h"
 6 #include "OutputFileStreamPipe.h"
 7 #include "RecordDossier.h"
 8 #include "StructureDescriptor.h"
 9 #include "InitializeStatic.C"

10 main()
11 {
12     enum FileType  { UNDEFINED, ASCII, BINARY, SWAPPED };
13     enum Status    { INVALID, VALID };
14     char BUF[512];

15     /** ASCII, binary, or binary byte-swapped representation? **/
16     FileType fileType = UNDEFINED;

17     while ( fileType == UNDEFINED ) {
18         cout << "Ascii, Binary, or Binary Byte-Swapped (A, B, or S) ? ";
19         cin >> BUF;
20         if ( BUF[0] == 'a' || BUF[0] == 'A' ) {
21             fileType = ASCII;
22         } else if ( BUF[0] == 'b' || BUF[0] == 'B' ) {
23             fileType = BINARY;
24         } else if ( BUF[0] == 's' || BUF[0] == 'S' ) {
25             fileType = SWAPPED;
26         } else {
27             cerr << "ERROR: invalid format type.\n";
28         }
29     }

30     /** Accept output file name and open output file **/
31     Status               fileStatus;
32     OutputFileStreamPipe *outFile;
33     PipeWriter           *pipeWriter;
```

```
34      do {
35          cout << "Please enter name of data file for output: ";
36          cin >> BUF;

37          outFile = new OutputFileStreamPipe( BUF );
38          if ( outFile->successfulOpen() ) {
39              fileStatus = VALID;
40              if ( fileType == ASCII ) {
41                  pipeWriter = new AsciiPipeWriter( outFile );
42              } else if ( fileType == BINARY ) {
43                  pipeWriter = new BinaryPipeWriter( outFile );
44              } else {
45                  pipeWriter = new ConversionPipeWriter( outFile );
46              }
47          } else {
48              cerr << "ERROR: unable to open specified file.\n";
49              fileStatus = INVALID;
50              delete outFile;
51          }
52      } while ( fileStatus == INVALID );

53      /** Class instance, pointers, and tag variable used to build SDDF packets **
54       ** In this program no field has > 2 dimensions; dimSizes set accordingly **/
55      Attributes              attributes;
56      StructureDescriptor     *structureP;
57      FieldDescriptor         *fieldP;
58      int                     tag = 1;
59      int                     dimSizes[2];

60      /** Stream Attribute **/
61      attributes.insert( "Creation Date", "Feb. 1, 1992" );
62      attributes.insert( "Machine", "Intel iPSC/860" );
63      pipeWriter->putAttributes( attributes );

64      /* Message Send structure */
65      attributes.clearEntries();
66      attributes.insert( "Event", "Message sent to one or more processors" );
67      structureP = new StructureDescriptor( "message send", attributes );

68          attributes.clearEntries();
69          attributes.insert( "Source", "Processor Sending Message" );
70          fieldP = new FieldDescriptor( "sourcePE", attributes, INTEGER, 0 );
71          structureP->insert( *fieldP );
72          delete fieldP;

73          attributes.clearEntries();
74          attributes.insert( "Destination", "Processor(s) Receiving Message" );
75          fieldP = new FieldDescriptor( "destinationPE", attributes, INTEGER, 1 );
76          structureP->insert( *fieldP );
77          delete fieldP;
```

```
78          attributes.clearEntries();
79          attributes.insert( "Size", "Message length in bytes" );
80          fieldP = new FieldDescriptor( "messageLength", attributes, INTEGER, 0 );
81          structureP->insert( *fieldP );
82          delete fieldP;

83          attributes.clearEntries();
84          attributes.insert( "Type", "Character denoting message type" );
85          fieldP = new FieldDescriptor( "messageType", attributes, CHARACTER, 0 );
86          structureP->insert( *fieldP );
87          delete fieldP;

88          attributes.clearEntries();
89          attributes.insert( "Description", "Timestamp" );
90          attributes.insert( "Units", "msec" );
91          fieldP = new FieldDescriptor( "timestamp", attributes, DOUBLE, 0 );
92          structureP->insert( *fieldP );
93          delete fieldP;

94      pipeWriter->putDescriptor( *structureP, tag );
95      RecordDossier *messageRecord = new RecordDossier( tag, *structureP );
96      Array *destinationPE = messageRecord->getArrayP( "destinationPE" );
97      delete structureP;
98      tag++;

99      /** Context Switch structure */
100     attributes.clearEntries();
101     structureP = new StructureDescriptor( "context switch", attributes );

102         attributes.clearEntries();
103         attributes.insert( "Description", "Clock" );
104         attributes.insert( "Units", "msec" );
105         fieldP = new FieldDescriptor( "timestamp", attributes, DOUBLE, 0 );
106         structureP->insert( *fieldP );
107         delete fieldP;

108         attributes.clearEntries();
109         fieldP = new FieldDescriptor( "processor", attributes, INTEGER, 0 );
110         structureP->insert( *fieldP );
111         delete fieldP;

112         fieldP = new FieldDescriptor( "procedure", attributes, CHARACTER, 1 );
113         structureP->insert( *fieldP );
114         delete fieldP;

115     pipeWriter->putDescriptor( *structureP, tag );
116     RecordDossier *contextSwitchRecord = new RecordDossier( tag, *structureP );
117     Array *procedure = contextSwitchRecord->getArrayP( "procedure" );
118     delete structureP;
119     tag++;
```

```
120     /** Procedure Call Summary Structure **/
121     attributes.clearEntries();
122     structureP = new StructureDescriptor( "procedure call summary", attributes );

123         fieldP = new FieldDescriptor( "data file", attributes, CHARACTER, 1 );
124         structureP->insert( *fieldP );
125         delete fieldP;

126         fieldP = new FieldDescriptor( "procedures", attributes, CHARACTER, 2 );
127         structureP->insert( *fieldP );
128         delete fieldP;

129         attributes.insert( "Dimension 0", "Procedure" );
130         attributes.insert( "Dimension 1", "Processor" );
131         attributes.insert( "Cell Value", "Count" );
132         fieldP = new FieldDescriptor( "count of calls", attributes, INTEGER, 2 );
133         structureP->insert( *fieldP );
134         delete fieldP;

135     pipeWriter->putDescriptor( *structureP, tag );
136     RecordDossier *procCallRecord = new RecordDossier( tag, *structureP );
137     Array *data_file = procCallRecord->getArrayP( "data file" );
138     Array *procedures = procCallRecord->getArrayP( "procedures" );
139     Array *count_of_calls = procCallRecord->getArrayP( "count of calls" );
140     delete structureP;
141     tag++;

142     /** Command Packet **/
143     pipeWriter->putCommand( 17 );

144     /** Stream Attribute **/
145     attributes.clearEntries();
146     attributes.insert( "Information", "End of Record Descriptors" );
147     pipeWriter->putAttributes( attributes );

148     /** Context Switch Data Packet **/
149     contextSwitchRecord->setValue( "timestamp", 100.15 );
150     contextSwitchRecord->setValue( "processor", 2 );
151     dimSizes[0] = 30;
152     procedure->setDimSizes( dimSizes );
153     procedure->setCellString( "run1" );
154     pipeWriter->putData( *contextSwitchRecord );

155     /** Message Send Data Packet **/
156     messageRecord->setValue( "sourcePE", 0 );
157     dimSizes[0] = 4;
158     destinationPE->setDimSizes( dimSizes );
159     destinationPE->setCellValue( 1, 0 );
160     destinationPE->setCellValue( 3, 1 );
161     destinationPE->setCellValue( 5, 2 );
162     destinationPE->setCellValue( 7, 3 );
```

```
163     messageRecord->setValue( "messageLength", 512 );
164     messageRecord->setValue( "messageType", 'B' );
165     messageRecord->setValue( "timestamp", 100.1 );
166     pipeWriter->putData( *messageRecord );

167     /** Message Send Data Packet **/
168     messageRecord->setValue( "sourcePE", 7 );
169     dimSizes[0] = 1;
170     destinationPE->setDimSizes( dimSizes );
171     destinationPE->setCellValue( 1, 0 );
172     messageRecord->setValue( "messageLength", 1012 );
173     messageRecord->setValue( "messageType", 'S' );
174     messageRecord->setValue( "timestamp", 102.15 );
175     pipeWriter->putData( *messageRecord );

176     /** Context Switch Data Packet - reuse procedure dimension size (=30) **/
177     contextSwitchRecord->setValue( "timestamp", 108 );
178     contextSwitchRecord->setValue( "processor", 4 );
179     procedure->setCellString( "printf" );
180     pipeWriter->putData( *contextSwitchRecord );

181     /** Message Send Data Packet **/
182     messageRecord->setValue( "sourcePE", 3 );
183     dimSizes[0] = 7;
184     destinationPE->setDimSizes( dimSizes );
185     destinationPE->setCellValue( 0, 0 );
186     destinationPE->setCellValue( 1, 1 );
187     destinationPE->setCellValue( 2, 2 );
188     destinationPE->setCellValue( 4, 3 );
189     destinationPE->setCellValue( 5, 4 );
190     destinationPE->setCellValue( 6, 5 );
191     destinationPE->setCellValue( 7, 6 );
192     messageRecord->setValue( "messageLength", 102 );
193     messageRecord->setValue( "messageType", 'B' );
194     messageRecord->setValue( "timestamp", 109 );
195     pipeWriter->putData( *messageRecord );

196     /** Context Switch Data Packet **/
197     contextSwitchRecord->setValue( "timestamp", 320 );
198     contextSwitchRecord->setValue( "processor", 5 );
199     procedure->setCellString( "scanf" );
200     pipeWriter->putData( *contextSwitchRecord );

201     /** Context Switch Data Packet **/
202     contextSwitchRecord->setValue( "timestamp", 412.77 );
203     contextSwitchRecord->setValue( "processor", 2 );
204     dimSizes[0] = 15;
205     procedure->setDimSizes( dimSizes );
206     procedure->setCellString( "printf" );
207     pipeWriter->putData( *contextSwitchRecord );
```

```
208     /** Procedure Call Summary Data Packet **/
209     dimSizes[0] = 30;
210     data_file->setDimSizes( dimSizes );
211     data_file->setCellString( "/mnt/data/testdata-1" );
212     dimSizes[0] = 3;
213     dimSizes[1] = 20;
214     procedures->setDimSizes( dimSizes );
215     procedures->setCellString( "compare()", 0 );
216     procedures->setCellString( "swap()", 1 );
217     procedures->setCellString( "error()", 2 );
218     dimSizes[0] = 3;
219     dimSizes[1] = 4;
220     count_of_calls->setDimSizes( dimSizes );
221     count_of_calls->setCellValue( 15, 0, 0 );
222     count_of_calls->setCellValue( 10, 0, 1 );
223     count_of_calls->setCellValue(  2, 0, 2 );
224     count_of_calls->setCellValue( 20, 0, 3 );
225     count_of_calls->setCellValue( 10, 1, 0 );
226     count_of_calls->setCellValue(  7, 1, 1 );
227     count_of_calls->setCellValue(  2, 1, 2 );
228     count_of_calls->setCellValue(  5, 1, 3 );
229     count_of_calls->setCellValue(  8, 2, 0 );
230     count_of_calls->setCellValue(  3, 2, 1 );
231     count_of_calls->setCellValue( 17, 2, 2 );
232     count_of_calls->setCellValue(  2, 2, 3 );
233     pipeWriter->putData( *procCallRecord );

234     /** Stream Attribute Packet **/
235     attributes.clearEntries();
236     attributes.insert( "information", "end of data" );
237     pipeWriter->putAttributes( attributes );

238     /** Close file **/
239     delete outFile;
240 }
```

E Sample SDDF File—ASCII Representation

This appendix contains a sample ASCII representation SDDF file demonstrating most of the data types and features provided by the SDDF specification.

```
SDDFA
/*
 * "Creation Date" "Feb. 1, 1992"
 * "Machine" "Intel iPSC/860"
 */ ;;

#1:
// "Event" "Message sent to one or more processors"
"message send" {
        // "Source" "Processor Sending Message"
        int     "sourcePE";
        // "Destination" "Processor(s) Receiving Message"
        int     "destinationPE"[];
        // "Size" "Message length in bytes"
        int     "messageLength";
        // "Type" "Character denoting message type"
        char    "messageType";
        // "Description" "Timestamp"
        // "Units" "msec"
        double  "timestamp";
};;

#2:
"context switch" {
        // "Description" "Clock"
        // "Units" "msec"
        double  "timestamp";
        int     "processor";
        char    "procedure"[];
};;

#3:
"procedure call summary" {
        char    "data file"[];
        char    "procedures"[][];
        // "Dimension 0" "Procedure"
        // "Dimension 1" "Processor"
        // "Cell Value" "Count"
        int     "count of calls"[][];
};;

%17: ;;

/*
 * "Information" "End of Record Descriptors"
 */ ;;
```

```
"context switch" { 100.150000, 2,
        [30] {
        "run1"
        } };;

"message send" { 0,
        [4] {
        1,
        3,
        5,
        7
        }, 512, B, 100.100000 };;

"message send" { 7,
        [1] {
        1
        }, 1012, S, 102.150000 };;

"context switch" { 108.000000, 4,
        [30] {
        "printf"
        } };;

"message send" { 3,
        [7] {
        0,
        1,
        2,
        4,
        5,
        6,
        7
        }, 102, B, 109.000000 };;

"context switch" { 320.000000, 5,
        [30] {
        "scanf"
        } };;

"context switch" { 412.770000, 2,
        [15] {
        "printf"
        } };;

"procedure call summary" {
        [30] {
        "/mnt/data/testdata-1"
        },
        [3][20] {
        "compare()",
```

```
        "swap()",
        "error()"
        },
        [3][4] {
        15,
        10,
        2,
        20,
        10,
        7,
        2,
        5,
        8,
        3,
        17,
        2
        } };;

/*
 * "information" "end of data"
 */ ;;
```

F Revision History

This appendix contains the revision history of the Pablo Self-Defining Data Format. Only changes to features covered in this manual are listed.

March 17, 1992

- Initial release.

Febrary 15, 1993

- Added support for input/output buffering to improve performance in the *InputFileStreamPipe* and *OutputFileStreamPipe* classes. The constructors for these classes now accept the buffer size to be used as an optional second argument. A default size is used if none is given.
- Updated to work with more recent C++ compilers.

May 6, 1993

- Added file headers specifying the file's SDDF representation type. SDDF files created with previous versions of the SDDF grammar specification must have headers prepended before they can be processed by the SDDF interface library routines in this release.
- Added support for binary byte-swapped representation and processing via the *ConversionPipeReader* and *ConversionPipeWriter* classes.
- Added the *InputFileStreamPipe::createPipeReader* method. Using this method the programmer need not know the SDDF representation of the file being read. Previous versions required the programmer to construct the appropriate *AsciiPipeReader* or *BinaryPipeReader* directly.

April 11, 1994

- Pablo is now a registered trademark of the Board of Trustees of the University of Illinois.
- Corrected line number reference to code fragment in section 4.10.
- Added support for zero dimension size in arrays. Ascii grammar updated to include non-negative integer production rule.

References

[1] NCSA. *NCSA HDF, Version 2.0.* University of Illinois at Urbana-Champaign, National Center for Supercomputing Applications, Feb. 1989.

[2] Reed, D. A., Olson, R. D., Aydt, R. A., Madhyastha, T. M., Birkett, T., Jensen, D. W., Nazief, B. A. A., and Totty, B. K. Scalable Performance Environments for Parallel Systems. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society Press.

[3] Rew, R. K. *netCDF User's Guide, Version 1.0.* Unidata Program Center, University Corporation for Atmospheric Research, Apr. 1989.

[4] The Picasso Research Group. *A Description of the Classes and Methods of the Pablo SDDF Interface Library.* University of Illinois at Urbana-Champaign, Department of Computer Science, May 1993.