

Daniel B. Michelson • Anders Henja

**A High Level Interface to the HDF5 File
Format**

Copyright 2002 by the Swedish Meteorological and Hydrological Institute (SMHI), Norrköping, Sweden.
All Rights Reserved

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its documentation, without fee, is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Swedish Meteorological and Hydrological Institute or SMHI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SMHI DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SMHI BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

\hline

Copyright Notice and Statement for NCSA Hierarchical Data Format (HDF) Software Library and Utilities NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities Copyright 1998, 1999, 2000 by the Board of Trustees of the University of Illinois. All rights reserved. Contributors: National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC), Lawrence Livermore National Laboratory (LLNL), Sandia National Laboratories (SNL), Los Alamos National Laboratory (LANL), Jean-loup Gailly and Mark Adler (gzip library). Redistribution and use in source and binary forms, with or without modification, are permitted for any purpose (including commercial purposes) provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or materials provided with the distribution.
3. In addition, redistributions of modified forms of the source or binary code must carry prominent notices stating that the original code was changed and the date of the change.
4. All publications or advertising materials mentioning features or use of this software are asked, but not required, to acknowledge that it was developed by the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign and to credit the contributors.
5. Neither the name of the University nor the names of the Contributors may be used to endorse or promote products derived from this software without specific prior written permission from the University or the Contributors.
6. THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY AND THE CONTRIBUTORS "AS IS" WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. In no event shall the University or the Contributors be liable for any damages suffered by the users arising out of the use of this software, even if advised of the possibility of such damage.

Contents

1	What's new in release 0.34?	1
1.1	Possibility to read/write references	1
1.2	Possibility to use HDF5 1.4 release	1
1.3	Miscellaneous	1
2	The W5 of HL-HDF	3
2.1	What?	3
2.2	Why?	3
2.3	Where?	4
2.4	When?	4
2.5	Who?	4
3	Compilation and Installation	5
3.1	Requirements	5
3.1.1	UNIX	5
3.1.2	Windows NT	5
3.1.3	Mac	5
3.1.4	VMS	5
3.2	Compilation	5
3.2.1	UNIX	6
3.2.2	Windows NT	6
3.3	Testing	6
3.3.1	UNIX	6
3.3.2	Windows NT	7
3.4	Installation	7
3.4.1	UNIX	7
3.4.2	Windows NT	7
3.5	Platform Notes	7
3.5.1	Red Hat Linux 6.2	7
3.5.2	Red Hat Linux 5.2	7
3.5.3	Sun	7
3.5.4	DEC Alpha	8
3.5.5	HP-UX	8
3.5.6	SGI	8
3.5.7	Cray T3E	8
3.5.8	Cray C90	8
3.5.9	Mac	8
3.5.10	Windows NT	8

4	Fundamentals	9
4.1	The Heirarchy	9
4.2	HL-HDF Building Blocks	10
4.2.1	Datatype	10
4.2.2	Attribute	10
4.2.3	Reference	10
4.2.4	Dataset	10
4.2.5	Group	10
4.2.6	Node	10
4.2.7	Scalar	10
4.2.8	Atomic	10
4.3	C Header Definitions	10
4.3.1	hid_t	11
4.3.2	herr_t	11
4.3.3	hsize_t	11
4.3.4	HL_Type	11
4.3.5	HL_DataType	11
4.3.6	HL_NodeMark	11
4.3.7	HL_Node	11
4.3.8	HL_NodeList	12
4.3.9	HL_CompoundTypeAttribute	12
4.3.10	HL_CompoundTypeDescription	12
5	Library Reference	13
5.1	General functions	13
5.1.1	initHlHdf	13
5.1.2	disableErrorReporting	13
5.1.3	enableErrorReporting	13
5.1.4	debugHlHdf	13
5.1.5	isHdf5File	13
5.1.6	openHlHdfFile	13
5.1.7	createHlHdfFile	14
5.1.8	closeHlHdfFile	14
5.1.9	getFixedType	14
5.1.10	translateCharToDatatype	14
5.1.11	getTypeNameString	14
5.1.12	getFormatNameString	14
5.1.13	getStringPadName	14
5.1.14	getStringCsetName	14
5.1.15	getStringCtypeName	15
5.1.16	whatSizeIsHdfFormat	15
5.1.17	isFormatSupported	15
5.1.18	newHL_Node	15
5.1.19	newHL_NodeList	15
5.1.20	freeHL_Node	15
5.1.21	freeHL_NodeList	15
5.1.22	newHL_Group	15
5.1.23	newHL_Attribute	15
5.1.24	newHL_Reference	16
5.1.25	newHL_Dataset	16
5.1.26	newHL_Datatype	16
5.1.27	newHL_CompoundTypeAttribute	16
5.1.28	newHL_CompoundTypeDescription	16

5.1.29	freeHL_CompoundTypeAttribute	16
5.1.30	freeHL_CompoundTypeDescription	16
5.1.31	addNode	16
5.1.32	getNode	17
5.1.33	setScalarValue	17
5.1.34	setArrayValue	17
5.1.35	extractParentChildName	17
5.1.36	commitDatatype	17
5.1.37	scanNodeList	17
5.1.38	findCompoundTypeDescription	17
5.1.39	scanCompoundTypeDescription	17
5.2	Display functions	18
5.3	Read functions	18
5.3.1	readHL_NodeListFrom	18
5.3.2	readHL_NodeList	18
5.3.3	selectNode	18
5.3.4	selectAllNodes	18
5.3.5	fetchMarkedNodes	18
5.3.6	fillAttributeNode	18
5.3.7	fillReferenceNode	18
5.3.8	fillDatasetNode	18
5.3.9	fillGroupNode	18
5.3.10	fillTypeNode	19
5.3.11	fillNodeWithData	19
5.3.12	buildTypeDescriptionFromTypeHid	19
5.3.13	locateNameForReference	19
5.4	Write functions	19
5.4.1	commitType	19
5.4.2	createStringType	19
5.4.3	setTypeSize	19
5.4.4	closeType	19
5.4.5	writeScalarDataAttribute	19
5.4.6	writeScalarDataAttribute_fmt	20
5.4.7	writeSimpleDataAttribute	20
5.4.8	writeSimpleDataAttribute_fmt	20
5.4.9	createSimpleDataset	20
5.4.10	createSimpleDataset_fmt	20
5.4.11	closeDataset	20
5.4.12	createCompoundType	20
5.4.13	addAttributeToCompoundType	20
5.4.14	addAttributeToCompoundType_fmt	21
5.4.15	addArrayToCompoundType	21
5.4.16	addArrayToCompoundType_fmt	21
5.4.17	createGroup	21
5.4.18	closeGroup	21
5.4.19	createReference	21
5.4.20	doWriteHdf5Attribute	21
5.4.21	doWriteHdf5Group	22
5.4.22	doWriteHdf5Dataset	22
5.4.23	doCommitHdf5Datatype	22
5.4.24	writeNodeList	22
5.5	Update functions	22
5.5.1	doAppendHdf5Attribute	22

5.5.2	doAppendHdf5Group	22
5.5.3	doAppendHdf5Dataset	22
5.5.4	updateNodeList	23
5.6	Deprecated	23
5.6.1	newGroup	23
5.6.2	newDataset	23
5.6.3	newNameListType	23
5.6.4	newAttribute	23
5.6.5	newCompoundAttribute	23
5.6.6	createCompoundFromType	23
5.6.7	addCompoundAttributeToType	23
5.6.8	addAttributeToGroup	23
5.6.9	addAttributeToDataset	24
5.6.10	freeCompoundAttribute	24
5.6.11	freeAttribute	24
5.6.12	freeNameListType	24
5.6.13	freeInternalDataset	24
5.6.14	freeDataset	24
5.6.15	addTypeToLocalGroup	24
5.6.16	addTypeToGlobalGroup	24
5.6.17	doesTypeExistInGlobalGroup	24
5.6.18	doesTypeExistInLocalGroup	24
5.6.19	removeTypeFromLocalGroup	25
5.6.20	removeTypeFromGlobalGroup	25
5.6.21	displayDataBuffer	25
5.6.22	displayCompoundDataset	25
5.6.23	displayCompoundAttributeDef	25
5.6.24	displayType	25
5.6.25	displayAttribute	25
5.6.26	displayDataset	25
5.6.27	displayGroup	26
5.6.28	readHIHdfFile	26
5.6.29	readHIHdfFileFrom	26
5.6.30	read_hlhdf_free	26
6	Creating your own HDF5 product	27
7	Example Programs	31
7.1	hlenc	31
7.2	hldec	31
7.3	hllist	31
8	Python Interface - PyHL	33
8.1	Compilation and installation	33
8.2	Create module _pyhl	33
8.3	Library Reference	33
8.3.1	nodelist	34
8.3.2	node	34
8.4	Examples	35
8.4.1	Writing a simple HDF5 file	35
8.4.2	Writing an HDF5 file containing a compound datatype	37
8.4.3	Reading a simple HDF5 file	38
8.4.4	Reading an HDF5 file containing a compound type	39

8.4.5	Reading an HDF5 file containing a compound type (alternative)	39
8.4.6	Creating a HDF5 image with a reference	40

Chapter 1

What's new in release 0.34?

1.1 Possibility to read/write references

The possibility to read and write references has been added.

1.2 Possibility to use HDF5 1.4 release

Since the HDF5 api has changed from release 1.4, HL-HDF has been modified to be possible to use with both HDF5 release 1.2 and 1.4.

1.3 Miscellaneous

Corrected both erroneous information and typos in this document.

Chapter 2

The W5 of HL-HDF

2.1 What?

HL-HDF is a high level interface to the Hierarchical Data Format, version 5, developed and maintained by the National Center for Supercomputing Applications (NCSA), at the University of Illinois at Urbana-Champaign. HDF5 is a file format designed for a maximum of flexibility and efficiency and it makes use of modern software technology. Briefly, HDF5 has the following characteristics:

- Platform independence. For example, an array of native floating point values written on one platform will be automatically identified, byte-swapped if necessary, and returned as an array of native floating point values on another platform.
- Built-in compression using the free ZLIB compression library. ZLIB is well-known as the compression used in the gzip package and it is robust and efficient.
- Flexible. HDF5 offers the ability to store virtually any kind of scientific data.

The HDF5 project URL is <http://www.hdfgroup.org/> and links are available to source code, software and copious documentation. HL-HDF is designed to focus on selected HDF5 functionality and make it available to users at a high level of abstraction, the idea being to make the management of their data easier. A strong effort has been made to ensure that this functionality, although a limited subset of HDF5 functionality, provides a general and flexible set of tools for managing arbitrary data. Like HDF5, HL-HDF is meant to be used on different computer platforms. In practise this means different flavours of UNIX and NT. HDF5 may work on other systems, like the Mac, DOS and VMS, but NCSA does not support them (yet) which means that HL-HDF does not support them either. This is not to say that they will not work on these platforms, however. Binary tools available with HDF5 will work with files written with HL-HDF. For example, *h5dump* can be used to determine the contents of an HDF5 file written with HL-HDF. A few test programs are included with HL-HDF which can read/write raw data to/from HDF5 files. These test programs may be useful to users who do not wish to write their own routines using the functionality available in HL-HDF, but would prefer to rely on a simpler encoder and decoder. These programs can also be used as examples of how to write routines using HL-HDF. HL-HDF is free software and it may be used by anyone according to SMHT's and NCSA's copyright statements contained in this document. Users are encouraged to report their feedback to the author, and to contribute to its development.

2.2 Why?

This software is designed to facilitate the management of scientific data from multiple sources. The integration of observations from various observational systems such as weather stations, satellites and radars is an area which is receiving increased attention. An increasing amount of work is also being carried out on integrating

such observations with information from numerical models, and in assimilating the observations into the models. Unfortunately, most types of data are stored in different file formats and little effort has been made to facilitate the exchange of data between disciplines such as meteorology, hydrology and oceanography. Since SMHI is the national agency responsible for operational activities in all three of these disciplines, it would obviously be beneficial to these operations if a rationalization of data management procedures can be realized. This is the reason why the HL-HDF software has been developed. Another important reason why HL-HDF has been developed is that it facilitates the management of multi-source data for pure research and development activities. This is due to the software's flexibility which provides a platform for managing virtually any variable and combination of variables imaginable. Due to HDF5's platform independent nature, its use can even be considered for exchange between organizations, either domestically or internationally. Its built-in compression is efficient which increases the potential amount of data available in archives and helps make them more useful.

2.3 Where?

HL-HDF has been developed for use in three general areas:

1. General purpose research and development.
2. Data management. HL-HDF can be used wherever there are requirements put on managing scientific data, whether it be with a small amount of data by a single person or with a comprehensive archive by a large organization.
3. Data exchange. HL-HDF can be used almost anywhere data exchange is required. This can be within an organization or between organizations, either domestically or internationally.

2.4 When?

HL-HDF has been developed during the first half of 2000 on a small budget as a one-off pilot project. This means that there is no ongoing project group and no official support. The objective has been to develop HL-HDF and then release it for anyone to use as he or she pleases. Feedback is naturally welcome to the e-mail address on this document's front page, and we hope to be able to incorporate improvements as best we can. The timing of HL-HDF has been fortunate. Had we gotten underway earlier, we probably would have chosen HDF4. Had we waited until later, then several of our applications may have chosen inferior file formats. It feels as though this work has been done in the right place at the right time.

2.5 Who?

Who can use HL-HDF? Anyone who works with scientific data can use HL-HDF, whether it be research and development with a limited amount of data or management of vast volumes of data in operational environments. Who has worked on this project? The programming has been performed by Anders Henja from Adcore, Norrköping. Daniel Michelson has coordinated the project. Mike Folk and Quincey Koziol of the HDF group at the NCSA have also been very helpful. An "ad hoc reference group" has followed the project's progress. This group consists of the following people:

Øystein Godøy	The Norwegian Meteorological Institute
Harri Hohti	Finnish Meteorological Institute
Otto Hyvärinen	Finnish Meteorological Institute
Pirkko Pylkkö	Finnish Meteorological Institute
Per Källberg	European Centre for Medium Range Weather Forecasting and SMHI
Hans Alexandersson	SMHI
Bengt Carlsson	SMHI
Adam Dybbroe	SMHI
Jörgen Sahlberg	SMHI

Chapter 3

Compilation and Installation

3.1 Requirements

The Hierarchical Data Format, version 5, must be built and accessible. Source code and prebuilt releases of HDF5 are available from the National Centre for Supercomputing Applications at [. Follow the documentation from NCSA if you plan on building HDF5 yourself. An extremely important requirement is that an ANSI-compliant C compiler be used. Some native compilers cannot handle ANSI C and HL-HDF will therefore not build.](#)

3.1.1 UNIX

A number of GNU tools are required, or at least highly recommended, in order to build HL-HDF. These tools are:

1. `gzip`(including `zlib`), version 1.1.0 or higher
2. `tar`
3. `make`, GNU Make version 3.7x or higher (or compatible) all of which are available from [. GNU C \(and Fortran\) compilers can also be retrieved from this site.](#)

In order for to work, the ZLIB compression library must be compiled and installed. ZLIB is available at [.](#)

3.1.2 Windows NT

The free WiZ package, available from [, or the proprietary WinZip package, available from \[, should be installed and accessible. If you choose to link in the pre-compiled HDF5 libs, then you'll have to use the Microsoft Visual C++ compiler, since this is what was used to build the HDF5 package.\]\(#\)](#)

3.1.3 Mac

No support yet.

3.1.4 VMS

No support yet.

3.2 Compilation

Make sure that all the requirements presented in the previous Section are met.

3.2.1 UNIX

The first step is to unpack the distribution. For the purposes of this documentation, the path `/usr/local/src` will be the root of the installation. Unpack the distribution with

```
/usr/local/src % tar xvzf hlhdf_r0.34.tgz
```

This will create a directory called `hlhdf` and the distribution will be placed in it. If the above arguments fail, then you have not used GNU tar. HL-HDF has a **configure** script to determine paths to compilers, headers and libraries. In short it tries to find everything HL-HDF needs to be built. Execute the **configure** script. The most relevant arguments are:

<code>--prefix=PATH</code>	Set the root of the installation path. Defaults to <code>'/usr/local/hlhdf'</code>
<code>--with-zlib=INC,LIB</code>	Use GNU ZLIB compression headers located at INC and library located at LIB
<code>--with-hdf5=INC,LIB</code>	Use the HDF5 headers located at INC and libraries located at LIB
<code>--with-python=yes no</code>	Configure in Python support. Default is yes . Enables building a Python interface.
<code>--with-fortran=yes no</code>	Configure with Fortran. Default is no . Useful if integrating with F77 code.

There are a few more arguments and they are listed by executing

```
/usr/local/src/hlhdf % ./configure -help
```

If **configure** fails, which is unlikely, then you may be in trouble. See Section platforms 3.5 for platform-specific notes. The bottom line is that you may have to make some manual adjustments to your configuration files. If configuration has been carried out without any problems then you're ready to build HL-HDF with:

```
/usr/local/src/hlhdf % make
```

This will generate the library `libhlhdf.a` located in the `/usr/local/src/hlhdf/hlhdf` directory.

3.2.2 Windows NT

Unpack the distribution using WiZ or WinZip. The following build instructions apply to the Microsoft Visual C++ 6.0 compiler.

1. Start a new project by selecting "File - New - Projects - Win32 Static Library". Add appropriate Project name (hlhdf) and Location in this same window. No precompiled headers or MFC support is needed.
2. Tools - Options - Directories. Make sure you add the path to the HDF5 header files.
3. Project - Add to Project - Files. Go to where the source and header files for HL-HDF are located and add them all.
4. Project - Settings - C++ -> C++ -> . Set appropriate warning level and optimization.
5. Build - Build hlhdf.lib

This should generate the file `hlhdf.lib` in the `Debug` directory.

3.3 Testing

3.3.1 UNIX

An optional testing of the HL-HDF library may be performed by compiling a test program located in the `/usr/local/src/hlhdf/test` directory. Simply move to this directory and type

```
/usr/local/src/hlhdf/test % make
```

which should build the test program `testRaveObject`. This program can be used to test read or test write an artificial image along with a number of different kinds of header parameters. To test the reading, execute

```
/usr/local/src/hlhdf/test % testRaveObject read
```

and an ASCII representation of the contents of `rave_image_file.hdf` will be written to **stdout**. To test writing, execute

```
/usr/local/src/hlhdf/test % testRaveObject write
```

and an ASCII representation of the contents of `rave_image_file.hdf` will be written to **stdout** and the file itself will be re-written. Alternatively, if `rave_image_file.hdf` doesn't exist, execute the test program with the `write` argument first to create the file, and then read it to examine its contents. If this test program works, then you can be confident that the HL-HDF library works! (The above use of "rave" in the test program and file refers to Radar Analysis and Visualization Environment software, which is freely available software maintained by SMHI).

3.3.2 Windows NT

Testing involves creating a new project in Microsoft Visual \Cpp{ }. This same strategy should be applied when building the *hlenc*, *hldec* and *hllist* binaries.

1. Start a new project by selecting "File - New - Projects - Win32 Console Application". Add appropriate *Project name* (test) and *Location* in this same window. Then select "An empty project".
2. Tools - Options - Directories. Add paths to the HL-HDF header files and the newly build library *hlhdf.lib*. Make sure the paths to the HDF5 headers and library are there as well.
3. Project - Add to Project - Files. Go to where *test_raveobject.c* is located and add it.
4. Project - Settings - \C{ }\Cpp{ }. Set appropriate warning level and optimization.
5. Project - Settings - Link - Object/library modules. Add *hlhdf.lib hdf5.lib zlib.lib* in this order to the beginning of this list.
6. Build - Build test.exe
7. Open a DOS console and change to the directory containing *test.exe*. Execute `test write` to create an HDF test file. Execute `test read` to query the contents of this file. If this works, then you can be confident that your HL-HDF library works.

3.4 Installation

3.4.1 UNIX

Execute

```
/usr/local/src/hlhdf % make install
```

and the header files, libraries, binaries, scripts and an MK-file will be installed to the *include*, *lib*, *bin* and *mkf* directories located under the path specified by the *prefix* variable which was used when HL-HDF was build. HL-HDF is complete when this has been carried out. For information on how to compile and install the Python interface, see Chapter ??.

3.4.2 Windows NT

A specific installation has not been defined. It is up to the user to place the headers, library, and binaries in appropriate locations.

3.5 Platform Notes

HL-HDF has been built on a number of systems, most of which are different flavours of UNIX. Unfortunately HDF5 is not available for any form of VMS or other arcane operating system such as DOS. What follows is a collection of platform-specific notes. *If the pre-compiled binaries are installed, the file mkf/hldef.mk has to be modified manually to point to the locations of the HDF5 installation, the ZLIB installation, the compiler, etc.*

3.5.1 Red Hat Linux 6.2

Platform:Linux 2.2.14-5.0 #1 Tue Mar 7 21:07:39 EST 2000 i686 unknown **C Compiler:** egcs-2.91.66 **Notes:** gcc should be at least version 2.8.1 when using HDF5 1.2.2. HL-HDF was developed using Red Hat 5.2 with gcc version 2.7.2.3 and HDF5 1.2.0.

3.5.2 Red Hat Linux 5.2

Platform: Linux 2.0.36 #1 Tue Oct 13 22:17:11 EDT 1998 **C Compiler:** gcc version 2.7.2.3 **Notes:** Development platform.

3.5.3 Sun

Platform: SunOS 5.7 Generic sun4u sparc SUNW,Ultra-5_10 **C Compiler:** Sun WorkShop Compiler C Version 4 **Fortran Compiler:** Found but version not verified **Notes:** Debugging platform (Purify).

3.5.4 DEC Alpha

Platform: OSF1 V4.0 878 alpha **C Compiler:** DEC C V5.6-071 on Digital UNIX V4.0 (Rev. 878) **Fortran Compiler:** Found but version not verified

3.5.5 HP-UX

Platform: HP-UX B.10.20 E 9000/778 **C Compiler:** HP, native C-compiler with ANSI support. **Fortran Compiler:** Not found, version not verified **Notes:** It is absolutely vital that the C compiler be ANSI-compliant. Some native HP compilers are not and this may cause the compilation of HL-HDF to fail. A PyHL binary distribution was not built on this platform.

3.5.6 SGI

Platform: IRIX64 6.5 07151439 IP27 **C Compiler:** MIPSpro Compilers: Version 7.2.1.3m **Notes:** When using the configure script, the linker options *-Bstatic* and *-Bdynamic* are valid. Compiling the `_pyhlmodule.so` module failed however. It might be necessary to remove the `\$(LD_FORCE_STATIC)` and `\$(LD_FORCE_SHARE)` arguments from the *LIBRARIES* variable in the PyHL *Makefile*. Otherwise, the SGI compiler is very sensitive to the cleanliness of the code and requires tons of arguments to shut up:

```
CFLAGS= -woff 1174,1429,1209,1196,1685 -woff 799,803,835 -Wl,-woff,47,-woff,84,-woff,85,-woff,134
```

Add these arguments to the end of your *CFLAGS* variable. This list may not be complete. **Fortran Compiler:** MIPSpro Compilers: Version 7.2.1.3m **Notes:** FMI is gratefully acknowledged for letting use one of their machines.

3.5.7 Cray T3E

Platform: sn6326 2.0.4.90 unicosmk CRAY T3E **C Compiler:** Cray Standard C Version 6.3.0.2 **Fortran Compiler:** Not found, not verified **Notes:** A PyHL binary distribution was not built on this platform.

3.5.8 Cray C90

Platform: sn4004 9.0.2.8 roo.13 CRAY C90 **C Compiler:** Cray Standard C Version 4.0.4.0 **Fortran Compiler:** Not found, not verified **Notes:** A PyHL binary distribution was not built on this platform.

3.5.9 Mac

Platform:C **Compiler:****Notes:**

3.5.10 Windows NT

Platform: Intel Pentium II with NT 4.00.1381 **C Compiler:** Microsoft Visual \Cpp{ } 6.0 **Notes:** In order to compile the *hlenc*, *hldec* and *hllist* binaries, the file *getopt.c* must be compiled in with the project(s). A PyHL binary distribution was not built on this platform.

Chapter 4

Fundamentals

This Chapter presents HL-HDF's building blocks so that the user will have a knowledge of the proper terminology prior to working hands-on with the library. To make the most of the functionality in HL-HDF, users should have a working knowledge of the C programming language. This documentation is designed to be complimentary to the official HDF5 documentation and users should refer to the official set for more detail on HDF5's internal mechanisms.

4.1 The Heirarchy

The "H" in HDF stands for "Heirarchical" and this describes how HDF files are structured. An HDF file can be likened to a file system. At the root of the file system is a period (".") or a slash ("/") and the file may consist of an arbitrary number of levels of data, like subdirectories in a file system. For example, if a NOAA satellite image containing several spectral bands of data are stored in this manner, one way of doing so could look like this:

```
.
/NOAA 14/
/NOAA 14/info
/NOAA 14/info/xsize
/NOAA 14/info/ysize
/NOAA 14/Channel 1/
/NOAA 14/Channel 2/
/NOAA 14/Channel 3/
```

where info is an object containing header information. The same strategy could be used to store several polar scans of weather radar data, for example. Alternatively, a numerical weather prediction model state could be represented in part using GRIB descriptors like this:

```
.
/Level 0/
/Level 0/Type 105/
/Level 0/Type 105/Parameter 11/
/Level 0/Type 102/
/Level 31/
/Level 31/Type 109/
/Level 31/Type 109/Parameter 11/
```

Or, why not a point from a weather station containing wind speed and direction values:

```
.
/WMO 02064/
/WMO 02064/dd/
/WMO 02064/ff/
/WMO 02036/
```

4.2 HL-HDF Building Blocks

HL-HDF provides a number of building blocks which are defined in detail in the header file *vhldf.h*.

4.2.1 Datatype

A *Datatype* is a data representation consisting of atomic data types such as a string, byte, integer, floating point value of a given word size, or in the form of a C *struct* containing combinations of atomic types. A *Datatype* is used to describe the characteristics of one's data, and a number of *Datatypes* may collectively constitute a header. Every *Datatype* is given a name which is stored in a string; this string is used to represent the *Datatype* in the HDF file.

4.2.2 Attribute

An *Attribute* contains a string used to identify it, an array with up to four dimensions, and a number of *Datatypes* describing that *Attribute*. An *Attribute* is an appropriate object for storing point values, for example, and storing time series of them is enabled in the *Attribute* object.

4.2.3 Reference

A *Reference* is basically a pointer to another object. A *Reference* can only be pointed to a *Datatype*, a *Dataset* or a *Group*. Using the *Reference* might be needed when generating HDF5 images, since a reference to a palette has to be inserted in the *Dataset* object.

4.2.4 Dataset

A *Dataset* is a higher level object and contains a string used to identify it, an optional array with between one and four dimensions, an array of *Datatypes*, and an array of *Attributes*. A *Dataset* is an appropriate object for storing profile (transect) or image data, and it can be used to store time series of a given variable.

4.2.5 Group

A *Group* is the highest level object and consists of a string used to identify it and an arbitrary combination of any of the *Datatype*, *Attribute*, *Dataset*, and *Group* building blocks. The root of any HDF5 file (denoted with "." or "/") is always a *Group*.

4.2.6 Node

A *Node* is a term used in the HL-HDF code to refer to any of the above mentioned building blocks in an HDF5 file. In other words, any given object in the heirarchy is a *Node*.

4.2.7 Scalar

A *Scalar* is an individual value.

4.2.8 Atomic

In HDF5 the predefined datatypes (for example 'int', 'short', ...) are referred to as *Atomic*, as opposed to the *Compound* datatypes which are a combination of *Atomic* datatypes.

4.3 C Header Definitions

The previous section presented the principles of HL-HDF building blocks. This section presents their actual names and their definitions, along with some fundamentals from HDF5 itself.

4.3.1 hid_t

This variable comes from HDF5 and is a type for managing references to nodes. Each node reference is represented as an integer and *hid_t* keeps track of them.

4.3.2 herr_t

This variable comes from HDF5 and is a type for handling error codes.

4.3.3 hsize_t

This variable comes from HDF5 and represents a native multiple-precision integer.

4.3.4 HL_Type

This is an enumeration variable designed to identify the type of a given node. *HL_Type* can be any of following possible values:

```
UNDEFINED_ID=-1
ATTRIBUTE_ID=0
GROUP_ID=1
DATASET_ID=2
TYPE_ID=3
REFERENCE_ID=4
```

4.3.5 HL_DataType

This is an enumeration variable designed to identify the type of data in a given node. *HL_DataType* can be any of the following possible values:

```
DTYPE_UNDEFINED_ID=-1
HL_SIMPLE=0
HL_ARRAY=1
```

When new nodes are initiated, they contain *HL_DataType=DTYPE_UNDEFINED*.

4.3.6 HL_NodeMark

This is an enumeration variable designed to keep track of the status of a given node. *HL_NodeMark* can be any of the following possible values:

```
NMARK_ORIGINAL=0
NMARK_CHANGED=1
NMARK_SELECT=2
```

A node with *HL_NodeMark=NMARK_CHANGED* can be used to mark that it has been modified. A node with *HL_NodeMark=NMARK_SELECT* is used to indicate that this node should be read when performing a fetch.

4.3.7 HL_Node

This is a single node and is defined in the following structure:

```
typedef struct HL_Node {
    HL_Type type;          /* the type of node */
    char name[256];       /* the node's name */
    int ndims;            /* the number of dimensions in the array */
    hsize_t dims[4];      /* the dimensions of each of ndims */
    unsigned char* data;  /* actual data (fixed type) */
    unsigned char* rawdata; /* actual raw data, machine dependent */
    char format[64];      /* the string representation of the data type */
}
```

```
hid_t typeId;          /* reference to HDF's internal type management */
size_t dSize;         /* size of one value in data (fixed type) */
size_t rdSize;        /* size of one value in raw data, machine dependent */
HL_DataType dataType; /* identifies whether data is single or an array */
hid_t hdfId;          /* like typeId: for internal use */
HL_NodeMark mark;     /* is this node marked? */
HL_CompoundTypeDescription* compoundDescription; /* a list of compound
type descriptions*/
} HL_Node;
```

4.3.8 HL_NodeList

This type is a list of nodes and is structured like this:

```
typedef struct {
    char filename[256]; /* a file string */
    char tmp_name[512]; /* temporary names for internal use */
    int nNodes;         /* the number of nodes in the list */
    int nAllocNodes;   /* the number of allocated nodes in the list; internal */
    HL_Node** nodes;   /* the nodes themselves */
} HL_NodeList;
```

4.3.9 HL_CompoundTypeAttribute

This type is designed to describe an individual node with a complicated structure, ie. one which consists of more than atomic data types. It contains all the information required to interpret the contents of the node:

```
typedef struct {
    char attrname[256]; /* the Attribute's name */
    size_t offset;     /* the offset to where the data begins */
    char format[256];  /* the string representation of the atomic data type */
    int ndims;         /* the number of dimensions in the array */
    size_t dims[4];    /* the dimensions of each of ndims */
} HL_CompoundTypeAttribute;
```

4.3.10 HL_CompoundTypeDescription

This type is a list of *HL_CompoundTypeAttributes*. The reason why it's called "Description" is that it acts more like meta data than actual data, since it's just a collection of other nodes which may contain data, and is therefore more of a description than anything else. It is structured like this:

```
typedef struct {
    char typename[256]; /* the list's name */
    unsigned long objno[2]; /* markers used to tag nodes in the list */
    size_t size;         /* size of this data type */
    int nAttrs;         /* the number of attributes in the list */
    int nAllocAttrs;    /* the number of allocated attributes */
    HL_CompoundTypeAttribute** attrs; /* the attributes themselves */
} HL_CompoundTypeDescription;
```

Chapter 5

Library Reference

What follows is a list of HL-HDF C functions, along with their arguments and descriptions on how to use them. The functions given in this section are those declared in the header files in the HL-HDF source. The functions are grouped according to what they are designed to do.

5.1 General functions

5.1.1 `initHIHdf`

void `initHIHdf()`

Initiates the HL-HDF functions. This call must be made before anything else is done. Returns nothing.

5.1.2 `disableErrorReporting`

void `disableErrorReporting()`

Deactivates HDF5 debugging. Returns nothing.

5.1.3 `enableErrorReporting`

void `enableErrorReporting()`

Activates HDF5 debugging. Returns nothing.

5.1.4 `debugHIHdf`

void `debugHIHdf(int flag)`

Sets the debug mode. *flag* can be 0 (no debugging), 1 (debug only HL-HDF), or 2 (debug HL-HDF and HDF5). Returns nothing.

5.1.5 `isHdf5File`

int `isHdf5File(const char* filename)`

Checks whether *filename* is an HDF5 file. Returns 1 if it is and 0 otherwise.

5.1.6 `openHIHdfFile`

hid_t `openHIHdfFile(const char* filename, const char* how)`

Opens an HDF5 file. Arguments: *filename*: String containing the file name. *how*: What mode that should be used for opening the file, can be 'r' (read only), 'w' (write only) or 'rw' (read and write). Returns the hid_t reference upon success otherwise -1.

5.1.7 createHlHdfFile

`hid_t createHlHdfFile(const char* filename)`

Creates an HDF5 file *filename*, if the file already exists it will be truncated. *filename* is the name of the file to be created. Returns the `hid_t` reference upon success otherwise -1.

5.1.8 closeHlHdfFile

`herr_t closeHlHdfFile(hid_t file_id)`

Closes the HDF5 file with the `hid_t` reference *file_id*. Returns a value greater or equal to 0 upon success otherwise a negative value.

5.1.9 getFixedType

`hid_t getFixedType(hid_t type)`

Translates from the datatype specified by *type* to a native datatype. Returns the native datatype `hid_t` upon success, or a negative value on failure.

5.1.10 translateCharToDatatype

`hid_t translateCharToDatatype(const char* dataType)`

Creates an HDF5 datatype `hid_t` from the string representation *dataType*. *dataType* can be one of: `char`, `schar`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `llong`, `ullong`, `float`, `double`, `hsize`, `hssize`, `herr` or `hbool`. Returns a value < 0 upon failure, otherwise a `hid_t` reference to the new type.

5.1.11 getTypeNameString

`char* getTypeNameString(hid_t type)`

Translates the HDF5 type *type* to an HDF5 string representation of the datatype. The returned string can be one of: `H5T_STD_I8BE`, `H5T_STD_I8LE`, `H5T_STD_I16BE`, `H5T_STD_I16LE`, `H5T_STD_I32BE`, `H5T_STD_I32LE`, `H5T_STD_I64BE`, `H5T_STD_I64LE`, `H5T_STD_U8BE`, `H5T_STD_U8LE`, `H5T_STD_U16BE`, `H5T_STD_U16LE`, `H5T_STD_U32BE`, `H5T_STD_U32LE`, `H5T_STD_U64BE`, `H5T_STD_U64LE`, `H5T_NATIVE_SCHAR`, `H5T_NATIVE_UCHAR`, `H5T_NATIVE_SHORT`, `H5T_NATIVE_USHORT`, `H5T_NATIVE_INT`, `H5T_NATIVE_UINT`, `H5T_NATIVE_LONG`, `H5T_NATIVE_ULONG`, `H5T_NATIVE_LLONG`, `H5T_NATIVE_ULLONG`, `H5T_IEEE_F32BE`, `H5T_IEEE_F32LE`, `H5T_IEEE_F64BE`, `H5T_IEEE_F64LE`, `H5T_NATIVE_FLOAT`, `H5T_NATIVE_DOUBLE`, `H5T_NATIVE_LDOUBLE`, `H5T_STRING` or `H5T_COMPOUND`. Returns the string representation upon success, otherwise NULL.

5.1.12 getFormatNameString

`char* getFormatNameString(hid_t type)`

Translates the HDF5 type *type* to a HL-HDF string representation of the datatype. The returned string can be one of *dataType* can be one of `char`, `schar`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `llong`, `ullong`, `float`, `double`, `hsize`, `hssize`, `herr`, `hbool`, `string` or `compound`. Returns the string representation upon success, otherwise NULL.

5.1.13 getStringPadName

`char* getStringPadName(hid_t type)`

Returns a string representation of the type *type*'s padding. The returned string can be one of `H5T_STR_NULLTERM`, `H5T_STR_NULLPAD`, `H5T_STR_SPACEPAD` or `ILLEGAL_STRPAD`. Returns the string representation upon success, otherwise NULL.

5.1.14 getStringCsetName

`char* getStringCsetName(hid_t type)`

Returns a string representation of the type *type*'s character set. The returned string can be one of `H5T_CSET_ASCII` or `UNKNOWN_CHARACTER_SET`. Returns the string representation upon success, otherwise NULL.

5.1.15 getStringCtypeName

char* **getStringCtypeName**(hid_t type)

Returns a string representation of the type *type*'s character type. The returned string can be one of H5T_C_S1, H5T_FORTRAN_S1 or UNKNOWN_CHARACTER_TYPE. Returns the string representation upon success, otherwise NULL.

5.1.16 whatSizeIsHdfFormat

int **whatSizeIsHdfFormat**(const char* format)

Calculates the size in bytes that the specified type takes. The attribute *format* can be one of char, schar, uchar, short, ushort, int, uint, long, ulong, llong, ullong, float, double, hsize, hssize, herr or hbool. Returns the size in bytes if successful or -1 in case of failure.

5.1.17 isFormatSupported

int **isFormatSupported**(const char* format)

Checks whether the string type *format* is recognized. *format* can be one of char, schar, uchar, short, ushort, int, uint, long, ulong, llong, ullong, float, double, hsize, hssize, herr or hbool. Returns 1 if the *format* is supported, otherwise 0.

5.1.18 newHL_Node

HL_Node* **newHL_Node**(const char* name)

Defines a new, empty node of undefined type. *name* is a string used to identify the node. Returns the node if successful or NULL upon failure.

5.1.19 newHL_NodeList

HL_NodeList* **newHL_NodeList**()

Creates an empty HL node list which can be filled with an arbitrary number of nodes. Returns the node list if successful or NULL upon failure.

5.1.20 freeHL_Node

void **freeHL_Node**(HL_Node* node)

Frees a node from memory. The node is given as the only argument. Returns nothing.

5.1.21 freeHL_NodeList

void **freeHL_NodeList**(HL_NodeList* nodelist)

Frees a complete node list from memory, along with all the nodes contained in it. The node list is given as the only argument. Returns nothing.

5.1.22 newHL_Group

HL_Node* **newHL_Group**(const char* name)

Creates an empty HL node of Group type. *name* is a string used to identify the node. Returns the node if successful or NULL upon failure.

5.1.23 newHL_Attribute

HL_Node* **newHL_Attribute**(const char* name)

Creates an empty HL node of Attribute type. *name* is a string used to identify the node. Returns the node if successful or NULL upon failure.

5.1.24 newHL_Reference

HL_Node* **newHL_Reference**(const char* name)

Creates an empty HL node of Reference type. *name* is a string used to identify the node. Returns the node if successful or *NULL* upon failure.

5.1.25 newHL_Dataset

HL_Node* **newHL_Dataset**(const char* name)

Creates an empty HL node of Dataset type. *name* is a string used to identify the node. Returns the node if successful or *NULL* upon failure.

5.1.26 newHL_Datatype

HL_Node* **newHL_Datatype**(const char* name)

Creates an empty HL node of Datatype type. *name* is a string used to identify the node. Returns the node if successful or *NULL* upon failure.

5.1.27 newHL_CompoundTypeAttribute

HL_CompoundTypeAttribute* **newHL_CompoundTypeAttribute**(char* attrname, size_t offset, \\char* format, int ndims, size_t* dims) Creates a compound Attribute node. This function is used to read nodes which are not simple atomic types. It is designed to hold the Attribute in the form of *unsigned char** along with information on how to interpret its contents. Arguments: *attrname*: String containing the Attribute's name. *offset*: The byte offset in the data where the Attribute's value starts. *format*: An atomic type, in character format, describing the Attribute, for example "short", or "double". *ndims*: Number of dimensions in the Attribute's array. *dims*: The dimensions of each of *ndims*. Returns the compound node if successful or *NULL* upon failure.

5.1.28 newHL_CompoundTypeDescription

HL_CompoundTypeDescription* **newHL_CompoundTypeDescription**()

Creates a list containing *HL_CompoundTypeAttributes*. Returns the compound type list if successful or *NULL* upon failure.

5.1.29 freeHL_CompoundTypeAttribute

```
\\begin{cfunccdesc}{void}{freeHL_CompoundTypeAttribute}{HL_CompoundTypeAttribute* attr}
```

Frees a given compound type attribute from memory. The only argument is the *HL_CompoundTypeAttribute* to be freed. Returns nothing.

5.1.30 freeHL_CompoundTypeDescription

```
void freeHL_CompoundTypeDescription(HL_CompoundTypeDescription* typelist)
```

Frees the compound type list, along with all its members, from memory. The only argument is the *HL_CompoundTypeDescription* to be freed. Returns nothing.

5.1.31 addNode

```
int addNode(HL_NodeList* nodelist, HL_Node* node)
```

Appends a node to (the end of) a node list. **Note:** If this operation is successful the responsibility for releasing the memory of the node *node* is taken by the nodelist, so do not release the *node* afterwards. Arguments: *nodelist*: The node list. *node*: The node to append to *nodelist*. Returns 1 if successful and 0 otherwise.

5.1.32 getNode

HL_Node* **getNode**(HL_NodeList* nodelist, const char* nodeName)

Provides a reference to a node from a node list. **Note:** A reference to the node is returned, so do not release the *node* when finished with the node. Arguments: *nodelist*: The node list. *nodeName*: A string identifying the node to extract. Returns (a reference to) the node if it is found, and *NULL* if not.

5.1.33 setScalarValue

int **setScalarValue**(HL_Node* node, size_t sz, unsigned char* value, const char* fmt, hid_t typid)

Writes a scalar value to a node. Scalar values are individual atomic words. Arguments: *node*: The node in which to write the value. *sz*: Size of the data type. *value*: The value to write. *fmt*: String representation of the data format, for example "short", "signed int" or "double". *typid*: Reference to used data type. Must be set manually if using a compound data type, otherwise set it to -1. Returns 1 if successful and 0 otherwise.

5.1.34 setArrayValue

int **setArrayValue**(HL_Node* node, size_t sz, int ndims, hsize_t* dims, unsigned char* value, const char* fmt, hid_t typid)

Writes an array to a node. Arguments: *node*: The node in which to write the array. *sz*: Size of the data type. *ndims*: The number of dimensions of the array, which may range from 0 to 4. *dims*: The dimensions of each of *ndims*. *value*: The array to write. *fmt*: String representation of the data format. *typid*: Reference to used data type. Must be set manually if using a compound data type, otherwise set it to -1. Returns 1 if successful and 0 otherwise.

5.1.35 extractParentChildName

int **extractParentChildName**(HL_Node* node, char* parent, char* child)

Separates the last node (the child) in a node name consisting of several nodes (the parent). For example, for a node name given as /group1/group2/group3, this function will set /group1/group2 as the parent and group3 as the child. Arguments: *node*: The node under scrutiny. *parent*: A string to hold the parent's node name. *child*: A string to hold the child's node name. Returns 1 if successful and 0 otherwise.

5.1.36 commitDatatype

int **commitDatatype**(HL_Node* node, hid_t testStruct_hid)

If a compound type has been created and there is a wish to have this node "named", then use this function for marking this node to be committed. See the HDF5 documentation for a more detailed description on what "committed" means. Arguments: *node*: A Datatype node to mark. *testStruct_hid*: The HDF5 hid_t reference to the datatype. Returns 1 if successful and 0 otherwise.

5.1.37 scanNodeList

void **scanNodeList**(HL_NodeList* nodelist)

Prints the names in a node list to the terminal. The only argument is the node list. Returns nothing.

5.1.38 findCompoundTypeDescription

HL_CompoundTypeDescription* **findCompoundTypeDescription**(HL_NodeList* nodelist, unsigned long objno0, unsigned long objno1) Searches a node list (*nodelist*) for all nodes with are identified by values *objno0* or *objno1*. Use this function to inquire whether an attribute's or dataset's type is "committed". Returns an *HL_CompoundTypeDescription* list if any nodes are found, otherwise *NULL*.

5.1.39 scanCompoundTypeDescription

void **scanCompoundTypeDescription**(HL_CompoundTypeDescription* typelist)

Prints to the terminal the names of all nodes in the *typelist* list of compound nodes. Returns nothing.

5.2 Display functions

5.3 Read functions

5.3.1 readHL_NodeListFrom

HL_NodeList* **readHL_NodeListFrom**(const char* filename, const char* fromPath)

Recursively reads the HDF5 file *filename* from the group *fromPath* and builds a list of nodes with corresponding names. I.e. no data will be read at this step, just the nodetypes and names will be determined. Returns an HL_NodeList pointer upon success, otherwise NULL.

5.3.2 readHL_NodeList

HL_NodeList* **readHL_NodeList**(const char* filename)

Recursively read the HDF5 file *filename* from the root group and builds a list of nodes with corresponding names. I.e. no data will be read at this step, just the nodetypes and names will be determined. Returns an HL_NodeList pointer upon success, otherwise NULL.

5.3.3 selectNode

int **selectNode**(HL_NodeList* nodelist, const char* name)

Marks the node with name *name* in the *nodelist* for retrieval. Returns 1 upon success, otherwise 0.

5.3.4 selectAllNodes

int **selectAllNodes**(HL_NodeList* nodelist)

Marks all nodes in the *nodelist* for retrieval. Returns 1 upon success, otherwise 0.

5.3.5 fetchMarkedNodes

int **fetchMarkedNodes**(HL_NodeList* nodelist)

Reads all nodes in the *nodelist* that has been marked for retrieval. Returns 1 upon success, otherwise 0.

5.3.6 fillAttributeNode

int **fillAttributeNode**(hid_t file_id, HL_Node* node)

Fills the attribute node *node* with data and dimensions from the file referenced by *file_id*. Returns 1 upon success, otherwise 0.

5.3.7 fillReferenceNode

int **fillReferenceNode**(hid_t file_id, HL_Node* node)

Fills the reference node *node* with data from the file referenced by *file_id*. The data field in the *node* will be filled with a string that specifies the name of the referenced *node*. Returns 1 upon success, otherwise 0.

5.3.8 fillDatasetNode

int **fillDatasetNode**(hid_t file_id, HL_Node* node)

Fills the dataset node *node* with data and dimensions from the file referenced by *file_id*. Returns 1 upon success, otherwise 0.

5.3.9 fillGroupNode

int **fillGroupNode**(hid_t file_id, HL_Node* node)

Fills the group node *node* with data from the file referenced by *file_id*. Returns 1 upon success, otherwise 0.

5.3.10 fillTypeNode

int **fillTypeNode**(hid_t file_id, HL_Node* node)

Fills the type node *node* with data from the file referenced by *file_id*. Returns 1 upon success, otherwise 0.

5.3.11 fillNodeWithData

int **fillNodeWithData**(hid_t file_id, HL_Node* node)

Fills the node *node* with data from the file referenced by *file_id*. Returns 1 upon success, otherwise 0.

5.3.12 buildTypeDescriptionFromTypeHid

HL_CompoundTypeDescription* **buildTypeDescriptionFromTypeHid**(hid_t type_id)

Builds a compound type description from the type *type_id* reference. Returns a HL_CompoundTypeDescription pointer upon success, otherwise NULL.

5.3.13 locateNameForReference

char* **locateNameForReference**(hid_t file_id, hobj_ref_t* ref)

This is a helper function for locating the name of an object that is referenced by *ref* in the file *file_id*. Returns a pointer to a string upon success, otherwise NULL.

5.4 Write functions

5.4.1 commitType

herr_t **commitType**(hid_t loc_id, const char* name, hid_t type_id)

Commits a datatype. See the HDF5 documentation for more detailed descriptions on what "committed" means. Arguments: *loc_id*: Where should the datatype be placed. *name*: What should the datatype be called. *type_id*: The hid_t reference to the datatype. Returns a negative value upon failure, otherwise the operation was successful.

5.4.2 createStringType

hid_t **createStringType**(size_t length)

Creates a HDF5 string type of length *length*. Returns a negative value upon failure, otherwise a hid_t reference to the datatype.

5.4.3 setTypeSize

herr_t **setTypeSize**(hid_t type_id, size_t theSize)

Changes the size of the datatype referenced by *type_id* to the size *theSize*. Returns a negative value upon failure, otherwise the operation was successful.

5.4.4 closeType

herr_t **closeType**(hid_t type_id)

Closes the datatype referenced by *type_id*. Returns a negative value upon failure, otherwise the operation was successful.

5.4.5 writeScalarDataAttribute

herr_t **writeScalarDataAttribute**(hid_t loc_id, hid_t type_id, const char* name, void* buf)

Writes a scalar value to an HDF5 file. Arguments: *loc_id*: The group or dataset the attribute should be written to. *type_id*: The datatype of the attribute. *name*: The name that should be used for the attribute. *buf*: The data that should be written. Returns 0 upon success, otherwise -1.

5.4.6 writeScalarDataAttribute_fmt

`herr_t writeScalarDataAttribute_fmt(hid_t loc_id, const char* fmt, const char* name, void* buf)`

Writes a scalar value to an HDF5 file. Arguments: *loc_id*: The group or dataset the attribute should be written to. *fmt*: A string describing the format of the datatype, e.g. char, short, ... *name*: The name that should be used for the attribute. *buf*: The data that should be written. Returns 0 upon success, otherwise -1.

5.4.7 writeSimpleDataAttribute

`herr_t writeSimpleDataAttribute(hid_t loc_id, hid_t type_id, const char* name, int ndims, hsize_t* dims, void* buf)`

Writes a simple data attribute value to an HDF5 file. Arguments: *loc_id*: The group or dataset the attribute should be written to. *type_id*: The datatype of the attribute. *name*: The name that should be used for the attribute. *ndims*: The rank of the data to be written, between 0-4. *dims*: The dimensions of the data, a pointer to *ndims* number of *hsize_t* values. *buf*: The data that should be written. Returns 0 upon success, otherwise -1.

5.4.8 writeSimpleDataAttribute_fmt

`herr_t writeSimpleDataAttribute_fmt(hid_t loc_id, const char* fmt, const char* name, int ndims, hsize_t* dims, void* buf)`

Writes a simple data attribute value to an HDF5 file. Arguments: *loc_id*: The group or dataset the attribute should be written to. *fmt*: A string describing the format of the datatype, e.g. char, short, ... *name*: The name that should be used for the attribute. *ndims*: The rank of the data to be written, between 0-4. *dims*: The dimensions of the data, a pointer to *ndims* number of *hsize_t* values. *buf*: The data that should be written. Returns 0 upon success, otherwise -1.

5.4.9 createSimpleDataset

`hid_t createSimpleDataset(hid_t loc_id, hid_t type_id, const char* name, int ndims, hsize_t* dims, void* buf, int compress)` Creates a dataset in an HDF5 file. Arguments: *loc_id*: The group the dataset should be created in. *type_id*: The datatype of the dataset. *name*: The name that should be used for the dataset. *ndims*: The rank of the data to be written. *dims*: The dimensions of the data, a pointer to *ndims* number of *hsize_t* values. *buf*: The data to be written in the dataset, if NULL, an empty dataset will be created. *compress*: The compression level on the dataset, between 0-9 where 0 is no compression and 9 is highest compression. Returns -1 on failure, otherwise a *hid_t* reference to the dataset.

5.4.10 createSimpleDataset_fmt

`hid_t createSimpleDataset_fmt(hid_t loc_id, const char* fmt, const char* name, int ndims, hsize_t* dims, void* buf, int compress)` Creates a dataset in an HDF5 file. Arguments: *loc_id*: The group the dataset should be created in. *fmt*: A string describing the format of the datatype, e.g. char, short, ... *name*: The name that should be used for the dataset. *ndims*: The rank of the data to be written. *dims*: The dimensions of the data, a pointer to *ndims* number of *hsize_t* values. *buf*: The data to be written in the dataset, if NULL, an empty dataset will be created. *compress*: The compression level on the dataset, between 0-9 where 0 is no compression and 9 is highest compression. Returns -1 on failure, otherwise a *hid_t* reference to the dataset.

5.4.11 closeDataset

`herr_t closeDataset(hid_t loc_id)`

Closes the dataset referenced by *loc_id*. Returns a negative value upon failure, otherwise the operation was successful.

5.4.12 createCompoundType

`hid_t createCompoundType(size_t size)`

Creates a compound type with the size *size*. Returns a negative value upon failure, otherwise a *hid_t* reference.

5.4.13 addAttributeToCompoundType

`herr_t addAttributeToCompoundType(hid_t loc_id, const char* name, size_t offset, hid_t type_id)`

Adds an scalar attribute to a compound type. Arguments: *loc_id*: The type the attribute should be appended to, *name*: The name of the attribute. *offset*: At what offset in the data does this attribute begin. *type_id*: The datatype of the attribute. Returns a negative value upon failure, otherwise the operation was successful.

5.4.14 addAttributeToCompoundType_fmt

`herr_t addAttributeToCompoundType_fmt(hid_t loc_id, const char* name, size_t offset, const char* fmt)`

Adds an scalar attribute to a compound type. Arguments: *loc_id*: The type the attribute should be appended to. *name*: The name of the attribute. *offset*: At what offset in the data does this attribute begin. *fmt*: A string describing the format of the datatype, e.g. char, short, ... Returns a negative value upon failure, otherwise the operation was successful.

5.4.15 addArrayType

`herr_t addArrayType(hid_t loc_id, const char* name, size_t offset, int ndims, size_t* dims, hid_t type_id)`

Adds an array attribute to a compound type. Arguments: *loc_id*: The type the attribute should be appended to. *name*: The name of the attribute. *offset*: At what offset in the data does this attribute begin. *ndims*: The rank of the data to be written, between 0-4. *dims*: The dimensions of the data, a pointer to *ndims* number of `hsize_t` values. *type_id*: The datatype of the attribute. Returns a negative value upon failure, otherwise the operation was successful.

5.4.16 addArrayType_fmt

`herr_t addArrayType_fmt(hid_t loc_id, const char* name, size_t offset, int ndims, size_t* dims, const char* fmt)` Adds an array attribute to a compound type. Arguments: *loc_id*: The type the attribute should be appended to. *name*: The name of the attribute. *offset*: At what offset in the data does this attribute begin. *ndims*: The rank of the data to be written, between 0-4. *dims*: The dimensions of the data, a pointer to *ndims* number of `hsize_t` values. *fmt*: A string describing the format of the datatype, e.g. char, short, ... Returns a negative value upon failure, otherwise the operation was successful.

5.4.17 createGroup

`hid_t createGroup(hid_t loc_id, const char* groupname, const char* comment)`

Creates a group in an HDF5 file. Arguments: *loc_id*: The group or file reference the group should be written to. *groupname*: The name of the group to be written. *comment*: A comment of the group, if NULL, no comment will be added to the group. Returns a negative value on failure, otherwise a `hid_t` reference.

5.4.18 closeGroup

`herr_t closeGroup(hid_t loc_id)`

Closes a group referenced by *loc_id*. Returns a negative value upon failure, otherwise the operation was successful.

5.4.19 createReference

`herr_t createReference(hid_t loc_id, hid_t file_id, const char* name, const char* targetname)`

Creates a reference from *name* in object *loc_id* to the object with the name *targetname* which must be a complete path in the file *file_id*. For example, to create a reference from **PALETTE** to the *dataset* /GRP/PALETTE one should write `createReference(loc_id, file_id, "PALETTE", "/GRP/PALETTE")`. *loc_id*: The object where the reference *name* should be created. *file_id*: The file the reference should be created in. *name*: The name of the reference to be created. *targetname*: The referenced object, must be a complete path. **Note that the referenced object must always be created before creating a reference to it.** Returns 0 upon success, otherwise -1

5.4.20 doWriteHdf5Attribute

`int doWriteHdf5Attribute(hid_t rootGrp, HL_Node* parentNode, char* parentName, HL_Node* childNode, char* childName)` Writes an `HL_Node` attribute to an HDF5 file. Arguments: *rootGrp*: The root group of the file. *parentNode*: The parent node of the attribute to be written. *parentName*: The name of the parent node. *childNode*: The node to be written. *childName*: The attribute's name. Returns 1 upon success, otherwise 0.

5.4.21 doWriteHdf5Group

int **doWriteHdf5Group**(hid_t rootGrp, HL_Node* parentNode, char* parentName, HL_Node* childNode, char* childName) Writes an HL_Node group to an HDF5 file. Arguments: *rootGrp*: The root group of the file. *parentNode*: The parent node of the group to be written. *parentName*: The name of the parent node. *childNode*: The node to be written. *childName*: The group's name. Returns 1 upon success, otherwise 0.

5.4.22 doWriteHdf5Dataset

int **doWriteHdf5Dataset**(hid_t rootGrp, HL_Node* parentNode, char* parentName, HL_Node* childNode, char* childName, int doCompress) Writes an HL_Node dataset to an HDF5 file. Arguments: *rootGrp*: The root group of the file. *parentNode*: The parent node of the dataset to be written. *parentName*: The name of the parent node. *childNode*: The node to be written. *childName*: The dataset's name. *doCompress*: The compression level on the dataset, between 0-9 where 0 is no compression and 9 is highest compression. Returns 1 upon success, otherwise 0.

5.4.23 doCommitHdf5Datatype

int **doCommitHdf5Datatype**(hid_t loc_id, HL_Node* parentNode, char* parentName, HL_Node* childNode, char* childName) Creates a "committed" datatype in the HDF5 file. Arguments: *rootGrp*: The root group of the file. *parentNode*: The parent node of the datatype to be written. *parentName*: The name of the parent node. *childNode*: The node to be written. *childName*: The datatype's name. Returns 1 upon success, otherwise 0.

5.4.24 writeNodeList

int **writeNodeList**(HL_NodeList* nodelist, int doCompress)

Writes a nodelist in HDF5 format. Arguments: *nodelist*: The nodelist to be written. *doCompress*: The compression level that should be used on the datasets, between 0-9 where 0 is no compression and 9 is highest compression. Returns 1 upon success, otherwise 0.

5.5 Update functions

As of release 0.30 some update functions have been added to HL-HDF. This allows the user to append data to an already existing HDF5 file. **Note:** Currently HDF5 does not support the possibility that the disk is full when writing a file which means that if one is unlucky the HDF5 file will be corrupt. To avoid this, estimate if it is likely that the file can be updated before attempting to update it.

5.5.1 doAppendHdf5Attribute

int **doAppendHdf5Attribute**(hid_t file_id, HL_Node* parentNode, char* parentName, HL_Node* childNode, char* childName) Appends an "attribute" node to the data structure. Arguments: *file_id*: The file reference. *parentNode*: The parent node of the datatype to be written. *parentName*: The name of the parent node. *childNode*: The node to be written. *childName*: The datatype's name. Returns 1 upon success, otherwise 0.

5.5.2 doAppendHdf5Group

int **doAppendHdf5Group**(hid_t file_id, HL_Node* parentNode, char* parentName, HL_Node* childNode, char* childName) Appends a "group" node to the data structure. Arguments: *file_id*: The file reference. *parentNode*: The parent node of the datatype to be written. *parentName*: The name of the parent node. *childNode*: The node to be written. *childName*: The datatype's name. Returns 1 upon success, otherwise 0.

5.5.3 doAppendHdf5Dataset

int **doAppendHdf5Dataset**(hid_t file_id, HL_Node* parentNode, char* parentName, HL_Node* childNode, char* childName) Appends a "dataset" node to the data structure. Arguments: *file_id*: The file reference. *parentNode*: The parent node of the datatype to be written. *parentName*: The name of the parent node. *childNode*: The node to be written. *childName*: The datatype's name. Returns 1 upon success, otherwise 0.

5.5.4 updateNodeList

int **updateNodeList**(HL_NodeList* nodelist, int doCompress) Updates the nodelist by calling the appropriate update function for each newly created node in the nodelist. Arguments: *nodelist*: The nodelist to be updated *doCompress*: The compression level that should be used on the datasets, between 0-9 where 0 is no compression and 9 is highest compression. Returns 1 upon success, otherwise 0.

5.6 Deprecated

Several functions are deprecated and are only provided for backward compatibility with alpha versions of this software which are actually being used. Avoid using these functions, since they will probably be removed in a future release.

5.6.1 newGroup

NameListGroup_t* **newGroup**(NameListGroup_t* parentGroup, const char* name)

Creates a new group named *name* and attaches this group to the *parentGroup*. If the *parentGroup* is NULL, then the created group will be the root group. Returns the new group upon success or NULL upon failure.

5.6.2 newDataset

NameListDataset_t* **newDataset**(NameListGroup_t* parentGroup, const char* name)

Creates a new dataset named *name* and attaches this dataset to the *parentGroup*. Returns the new dataset upon success or NULL upon failure.

5.6.3 newNameListType

NameListType_t* **newNameListType**()

Creates a new type object. Returns the allocated type upon success or NULL upon failure.

5.6.4 newAttribute

NameListAttribute_t* **newAttribute**(const char* name)

Creates a new attribute with the name *name*, if *name* is NULL, then the attribute will be nameless. Returns the allocated attribute upon success or NULL upon failure.

5.6.5 newCompoundAttribute

CompoundAttributeDef_t* **newCompoundAttribute**(const char* name)

Creates a new compound attribute definition with the name *name*, if *name* is NULL, then the attribute will be nameless. Returns the allocated compound attribute definition upon success or NULL upon failure.

5.6.6 createCompoundFromType

CompoundAttributeDef_t* **createCompoundFromType**(NameListType_t* inType, char* name)

Translates an NameListType_t instance to a compound attribute definition instance and then gives the compound attribute definition the name *name*. Returns the allocated compound attribute definition upon success or NULL upon failure.

5.6.7 addCompoundAttributeToType

herr_t **addCompoundAttributeToType**(NameListType_t* newType, CompoundAttributeDef_t* compoundAttr)

Adds the compound attribute definition *compoundAttr* to the name list type *newType*. Returns a value ≥ 0 upon success, otherwise -1.

5.6.8 addAttributeToGroup

herr_t **addAttributeToGroup**(NameListGroup_t* group, NameListAttribute_t* attr)

Adds the attribute *attr* to the group *group*. Returns 0 upon success otherwise -1.

5.6.9 addAttributeToDataset

`herr_t addAttributeToDataset(NameListDataset_t* dset, NameListAttribute_t* attr)`
Adds the attribute *attr* to the dataset *dset*. Returns 0 upon success otherwise -1.

5.6.10 freeCompoundAttribute

`void freeCompoundAttribute(CompoundAttributeDef_t* attr)`
Deallocates the compound attribute definition *attr*. Returns nothing.

5.6.11 freeAttribute

`void freeAttribute(NameListAttribute_t* attr)`
Deallocates the attribute *attr*. Returns nothing.

5.6.12 freeNameListType

`void freeNameListType(NameListType_t* type)`
Deallocates the name list type *type*. Returns nothing.

5.6.13 freeInternalDataset

`void freeInternalDataset(NameListDataset_t* dset)`
Deallocates the internals for the dataset *dset*. Returns nothing.

5.6.14 freeDataset

`void freeDataset(NameListDataset_t* dset)`
Deallocates the dataset *dset*. Returns nothing.

5.6.15 addTypeToLocalGroup

`herr_t addTypeToLocalGroup(NameListGroup_t* group, NameListType_t* type)`
Adds the type *type* to the local list of types in the group *group*. Returns a value ≥ 0 upon success otherwise -1.

5.6.16 addTypeToGlobalGroup

`herr_t addTypeToGlobalGroup(NameListGroup_t* group, NameListType_t* type)`
Adds the type *type* to the global list of types in the group *group*. Returns a value ≥ 0 upon success otherwise -1.

5.6.17 doesTypeExistInGlobalGroup

`int doesTypeExistInGlobalGroup(NameListGroup_t* grp, unsigned long* objno)`
Searches the global list of types in the group *grp* if any occurrence of the *objno* exists. Arguments: *grp*: The group that should be searched in. *objno*: An list of two unsigned long's. Returns the index number in the global list if an occurrence was found otherwise -1.

5.6.18 doesTypeExistInLocalGroup

`int doesTypeExistInLocalGroup(NameListGroup_t* grp, unsigned long* objno)`
Searches the local list of types in the group *grp* if any occurrence of the *objno* exists. Arguments: *grp*: The group that should be searched in. *objno*: A list of two unsigned long's. Returns the index number in the local list if an occurrence was found otherwise -1.

5.6.19 removeTypeFromLocalGroup

NameListType_t* **removeTypeFromLocalGroup**(NameListGroup_t* group,unsigned long* objno)

Removes the type with a matching *objno* from the group *group*'s list of local types and returns the type. Arguments: *group*: The group that should be searched. *objno*: A list of two unsigned long's. Returns the type with a matching object number if it was found, otherwise NULL

5.6.20 removeTypeFromGlobalGroup

NameListType_t* **removeTypeFromGlobalGroup**(NameListGroup_t* group,unsigned long* objno)

Removes the type with a matching *objno* from the group *group*'s list of global types and returns the type. Arguments: *group*: The group that should be searched. *objno*: A list of two unsigned long's. Returns the type with a matching object number if it was found, otherwise NULL

5.6.21 displayDataBuffer

void **displayDataBuffer**(unsigned char* data, const char*fmt, int ndims, hsize_t* dims, size_t typeSize, int offs, int addNewline)

Displays the data in a format similar to the one produced when using **h5dump** distributed with the HDF5 distribution. Arguments: *data*: A pointer to the data. *fmt*: The hlhd string representation of the dataformat. *ndims*: The rank of the data. *dims*: The dimensions of the data. *typeSize*: The size of each value. *offs*: The number of blanks that should be padded before the data. *addNewline*: If a linebreak should be added or not, 1 means add linebreak. Returns nothing.

5.6.22 displayCompoundDataset

void **displayCompoundDataset**(unsigned char* data,NameListType_t* type,int ndims, hsize_t* dims, int offs)

Displays a compound dataset in a format similar to the one produced when using **h5dump** distributed with the HDF5 distribution. Arguments: *data*: The data pointer. *type*: The compound type definition. *ndims*: The rank of the data. *dims*: The dimensions of the data. *offs*: The number of blanks that should be padded before the data. Returns nothing

5.6.23 displayCompoundAttributeDef

void **displayCompoundAttributeDef**(CompoundAttributeDef_t* def,int offs)

Displays one attribute in a compound attribute in a format similar to the one produced when using **h5dump** distributed with the HDF5 distribution. Arguments: *def*: The compound attribute definition. *offs*: The number of blanks that should be padded before the data. Returns nothing

5.6.24 displayType

void **displayType**(NameListType_t* type, int offs)

Displays one datatype in a format similar to the one produced when using **h5dump** distributed with the HDF5 distribution. Arguments: *type*: The datatype to display. *offs*: The number of blanks that should be padded before the data. Returns nothing.

5.6.25 displayAttribute

void **displayAttribute**(NameListAttribute_t* attr,int offs)

Displays one attribute in a format similar to the one produced when using **h5dump** distributed with the HDF5 distribution. Arguments: *attr*: The attribute to display. *offs*: The number of blanks that should be padded before the data. Returns nothing.

5.6.26 displayDataset

void **displayDataset**(NameListDataset_t* dset, int offs)

Displays one dataset in a format similar to the one produced when using **h5dump** distributed with the HDF5 distribution. Arguments: *dset*: The dataset to display. *offs*: The number of blanks that should be padded before the data. Returns nothing.

5.6.27 displayGroup

void **displayGroup**(NameListGroup_t* grp,int offs)

Displays one group in a format similar to the one produced when using **h5dump** distributed with the HDF5 distribution. This function will recursively go through all sub-groups belonging to this group. Arguments: *grp*: The group to display. *offs*: The number of blanks that should be padded before the data. Returns nothing.

5.6.28 readHIHdfFile

NameListGroup_t* **readHIHdfFile**(const char* filename)

Recursively reads a complete HDF5 file with name *filename* and builds a complete tree structure. Returns a pointer to a NameListGroup_t instance upon success, otherwise NULL.

5.6.29 readHIHdfFileFrom

NameListGroup_t* **readHIHdfFileFrom**(const char* filename, const char* from)

Recursively reads an HDF5 file with name *filename* from the group *from* and builds a complete tree structure. Returns a pointer to a NameListGroup_t instance upon success, otherwise NULL.

5.6.30 read_hlhdf_free

void **read_hlhdf_free**(NameListGroup_t* group)

Frees a HDF5 tree structure that has been read by using either **readHIHdfFile** or **readHIHdfFileFrom**. Be aware that this function must be used if one of the two functions above was used since it knows how the tree structure was built. Returns nothing.

Chapter 6

Creating your own HDF5 product

When creating your own HDF5 product, there are two header files that should be included, **read_vhllhdf.h** and **write_vhllhdf.h**. When compiling a binary, there are three libraries that must be linked in; these are **libhlhdf.a**, **libhdf5.a** and **libz.a**. It is also possible to link the shared library **libhdf5.so** instead of **libhdf5.a**. The HL-HDF package was installed with a **hldef.mk** file that can be included in your own Makefile in order to get the correct paths to both the zlib and the hdf5 library. It also contains information on which `\C{}`-compiler the HL-HDF package was compiled with and some other goodies. A simple Makefile could look like this:

```
include /usr/local/hlhdf/mkf/hldef.mk
HLHDF_INCDIR = -I/usr/local/hlhdf/include
HLHDF_LIBDIR = -L/usr/local/hlhdf/lib
CFLAGS = $(OPTS) $(DEFS) -I. $(ZLIB_INCDIR) $(HDF5_INCDIR) \
    $(HLHDF_INCDIR)
LDLDFLAGS = -L. $(ZLIB_LIBDIR) $(HDF5_LIBDIR) $(HLHDF_LIBDIR)
LIBS = -lhlhdf -lhdf5 -lz -lm
TARGET=myTestProgram
SOURCES=test_program.c
OBJECTS=$(SOURCES:.c=.o)
all: $(TARGET)
$(TARGET): $(OBJECTS)
    $(CC) -o $$@ $(LDLDFLAGS) $(OBJECTS) $(LIBS)

clean:
    @\rm -f *.o *~ so_locations core
distclean: clean
    @\rm -f $(TARGET)
distribution:
    @echo "Would bring the latest revision upto date"
install:
    @$ (HL_INSTALL) -f -o -C $(TARGET) ${MY_BIN_PATH}/$(TARGET)
```

Now, when the Makefile has been created, it might be a good idea to write your own HDF5 product. The following example will create a dataset with a two-dimensional array of integers, and two attributes connected to this dataset. It will also create a group containing one attribute.

```
#include <read_vhllhdf.h>
#include <write_vhllhdf.h>
int main(int argc, char** argv)
{
    HL_NodeList* aList=NULL;
    HL_Node* aNode=NULL;
    int* anArray=NULL;
    int anIntValue;
    float aFloatValue;
    hsize_t dims[]={10,10};
```

```

int npts=100;
int i;
initHLHdf(); /* Initialize the HL-HDF library */
debugHLHdf(2); /* Activate debugging */
if(!(aList = newHL_NodeList())) {
    fprintf(stderr,"Failed to allocate nodelist");
    goto fail;
}
if(!(anArray = malloc(sizeof(int)*npts)) {
    fprintf(stderr,"Failed to allocate memory for array.");
    goto fail;
}
for(i=0;i<npts;i++)
    anArray[i]=i;
addNode(aList,(aNode = newHL_Group("/group1")));
addNode(aList,(aNode = newHL_Attribute("/group1/attribute1")));
anInt Value=10;
setScalarValue(aNode,sizeof(anInt Value),(unsigned char*)&anInt Value,"int",-1);
addNode(aList,(aNode = newHL_Dataset("/dataset1")));
setArrayValue(aNode,sizeof(int),2,dims,(unsigned char*)anArray,"int",-1);
addNode(aList,(aNode = newHL_Attribute("/dataset1/attribute2")));
anInt Value=20;
setScalarValue(aNode,sizeof(anInt Value),(unsigned char*)&anInt Value,"int",-1);
addNode(aList,(aNode = newHL_Attribute("/dataset1/attribute3")));
aFloat Value=99.99;
setScalarValue(aNode,sizeof(aFloat Value),(unsigned char*)&aFloat Value,
    "float",-1);
strcpy(aList->filename,"written_hdffile.hdf");
writeNodeList(aList,6);
freeHL_NodeList(aList);
exit(0);
return 0; /* Won't come here */
fail:
freeHL_NodeList(aList);
exit(1);
return 1; /* Won't come here */
}

```

When you have created your own HDF5 product, it might be a good idea to create some code for reading this file and checking its contents.

```

#include <read_vhlhdf.h>
#include <write_vhlhdf.h>
int main(int argc, char** argv)
{
    HL_NodeList* aList=NULL;
    HL_Node* aNode=NULL;
    int* anArray=NULL;
    int anIntValue;
    float aFloatValue;
    int npts;
    int i;
    initHLHdf(); /* Initialize the HL-HDF library */
    debugHLHdf(2); /* Activate debugging */
    if(!(aList = readHL_NodeList("written_hdffile.hdf"))) {
        fprintf(stderr,"Failed to read nodelist\n");
        goto fail;
    }
    selectAllNodes(aList); /* Select everything for retrieval */
}

```

```

fetchMarkedNodes(aList);
if((aNode = getNode(aList, "/group1")))
    printf("%s exists\n", aNode->name);
if((aNode = getNode(aList, "/group1/attribute1"))) {
    memcpy(&anIntValue, aNode->data, aNode->dSize);
    printf("%s exists and have value %d\n", aNode->name, anIntValue);
}
if((aNode = getNode(aList, "/dataset1"))) {
    anArray = (int*)aNode->data;
    npts = 1;
    for(i=0; i<aNode->ndims; i++)
        npts*=aNode->dims[i];
    printf("%s exists and has the values:\n", aNode->name);
    for(i=0; i<npts; i++) {
        printf("%d ", anArray[i]);
        if((i%aNode->dims[0])==0) {
            printf("\n");
        }
    }
    printf("\n");
}
if((aNode = getNode(aList, "/dataset1/attribute2"))) {
    memcpy(&anIntValue, aNode->data, aNode->dSize);
    printf("%s exists and have the value %d\n", aNode->name, anIntValue);
}
if((aNode = getNode(aList, "/dataset1/attribute3"))) {
    memcpy(&aFloatValue, aNode->data, aNode->dSize);
    printf("%s exists and have the value %f\n", aNode->name, aFloatValue);
}
freeHL_NodeList(aList);
exit(0);
return 0; /* Never reached */
fail:
freeHL_NodeList(aList);
exit(1);
return 1; /* Never reached */
}

```


Chapter 7

Example Programs

Three example programs have been provided with HL-HDF. Two of them are modelled after the BUFR software developed and maintained by the EUMETNET Operational Programme for the Exchange of Weather Radar Information (OPERA). This software has two programs called *enbufr* and *debufr* used to encode and decode BUFR messages to/from an ASCII file containing header information and raw data in a binary file. The third example is modelled after a program called *griblist* developed by SMHI to query the contents of a GRIB file. GRIB and BUFR are format standards specified by the World Meteorological Organization.

7.1 hlenc

Encodes raw binary data in one file and an ASCII file containing header information, into an HDF5 file. **hlenc [-hdv] [-z compression] -i inputprefix -o outputfile[-h]** Prints a help text. [-d] Prints debugging information. [-v] Prints the version number. [-z compression] Sets the compression level, can be in the range 0 to 9 where 0 is no compression and 9 is the highest compression. -i *inputprefix* Specifies the prefix for the input files, the files that will be read are <inputprefix>.info and <inputprefix>.data. -o *outputfile* Specifies the name of the HDF5 file to be generated. The file with extension .info should have the following appearance: DATATYPE: [ATTRIBUTE or DATASET] FIELDNAME: [name of the field, e.g. '/attr1'] DATASIZE: [size of the datatype in bytes] DATAFORMAT: [string representation of the datatype, e.g. int] DIMS: [the dimension of the data embraced by [], e.g. [10,10]] The file with extension .data should contain raw binary data with native byte order.

7.2 hldec

Decodes an HDF5 file into a binary data file and an ASCII info file. **hldec [-hdv] -i inputfile -f fieldname -o outputprefix[-h]** Prints an help text. [-d] Prints debugging information. [-v] Prints the version number. -i *inputfile* Specifies the HDF5 file to be decoded. -f *fieldname* Specifies the fieldname to be decoded, e.g. '/dataset1'. -o *outputprefix* Specifies the prefix for the output files, the files that will be generated are <outputprefix>.info and <outputprefix>.data. The file with extension .info will get the following appearance: DATATYPE: [ATTRIBUTE or DATASET] FIELDNAME: [name of the field, e.g. '/attr1'] DATASIZE: [size of the datatype in bytes] DATAFORMAT: [string representation of the datatype, e.g. int] DIMS: [the dimension of the data embraced by [], e.g. [10,10]] The file with extension .data will be saved in byteformat with native byte order.

7.3 hllist

Lists the nodes in an HDF5 file. **hllist [-hdv] hdf5file[-h]** Prints a help text. [-d] Prints debugging information. [-v] Prints the version number. *hdf5file* Is the HDF5 file to be listed.

Chapter 8

Python Interface - PyHL

PyHL is just like the HL-HDF library in that it allows the user to work with HDF5 at a high level. PyHL is designed to work at the highest level of abstraction using the Python programming language, since Python allows the user to interact directly with HDF5 files. In fact, PyHL is nothing more than a wrapper around HL-HDF but with some additional functionality which is only available in very high level languages such as Python. Like HL-HDF, it is up to the user to define appropriate ways of representing data and using the building blocks available in PyHL to store the data in HDF5. (PyHL is pronounced "pile", which is an appropriate description of a heirarchy \ldots)

8.1 Compilation and installation

The Python programming language, version 1.5.2, is required along with the `\module{Numeric}` package. Python is found at the Corporation for National Research Initiatives at `\url{http://www.python.org/}` and `\module{Numeric}` is found at the Source Forge `\url{http://numpy.sourceforge.net/}`.

8.2 Create module `_pyhl`

If the configure script was not called with `--with-python=no` the `_pyhl` module should be compiled together with the rest of the code. If the configure script was called with `--with-python=no`, then the best thing is to rebuild the whole HL-HDF package (with `--with-python=yes`) and installation as described in Sections `\ref{compilation}` and `\ref{installation}`. **NOTE:** *Python version 1.5.2 is required to compile `_pyhl`; otherwise there will be unresolved symbols. Also, be aware that the `hdf5` library is linked dynamically which requires that the `LD_LIBRARY_PATH` contains the path to where `libhdf5.so` has been installed.*

8.3 Library Reference

This module defines the IO access for reading/writing HDF5 files. The module implements two classes for building an HDF5 file representation. The `nodelist` class implements a list that should represent the file itself. The `node` class represents the items in the HDF5 file. The `nodelist` contain several `node`'s for building the HDF5 file. You can use this interface to write Python programs that interfact with HDF5 files.

The module defines the following items:

is_file_hdf5(filename) Checks whether `filename` is an HDF5 file. Returns 1 if it is and 0 otherwise.

nodelist() Return a new instance of the `nodelist` class.

node(nodetype,nodename) Return a new instance of the `node`. The `nodetype` can be one of: `ATTRIBUTE_ID`, `DATASET_ID`, `GROUP_ID`, `TYPE_ID` and `REFERENCE_ID`. The `nodename` is the name of the node, for example `/group1/group2/dataset1.ATTRIBUTE_ID`

When creating a node and using this value, the node will become an attribute node.

DATASET_ID

When creating a node and using this value, the node will become a dataset node.

GROUP_ID

When creating a node and using this value, the node will become a group node.

TYPE_ID

When creating a node and using this value, the node will become a datatype node.

REFERENCE_ID

When creating a node and using this value, the node will become a reference node.

read_nodelist(filename[,from]) Read the HDF5 file *filename* and build a nodelist with all the names. That is the data will not be read, just the names. If a nodelist is built from a group lower down in the hierarchy, then *from* can be specified. If all goes well, the nodelist is returned, otherwise an exception is thrown. `\end{funcdesc}`

8.3.1 nodelist

nodelist instances have the following methods: **write(filename[,compression])** Write the instance to disk in HDF5 format. The default compression value is *6*. If another compression level is wanted, then the value can be between *0* for no compression and *9* for highest compression. **update([,compression])** Updates the instance. The default compression value is *6*. If another compression level is wanted, then the value can be between *0* for no compression and *9* for highest compression. **addNode(node)** Adds a node of class node to the end of the nodelist. **getNodeNames()** Returns a dictionary with all the nodelists' node names as keys and the integer values *ATTRIBUTE_ID*, *DATASET_ID*, *TYPE_ID*, *GROUP_ID* and *REFERENCE_ID* as items. **selectAll()** Marks all nodes in the nodelist for data retrieval. **selectNode(nodename)** Marks the node specified by *nodename* to be retrieved. **fetch()** Fetches all nodes in the selected nodelist. **getNode(nodename)** Return the node with name *nodename*.

8.3.2 node

A node has the following methods:

setScalarValue(itemSize,data,typename,lhid)

Sets a scalar value in the node instance. *itemSize* is used for specifying the size of the value in bytes. It is not necessary to specify unless a compound type is set. *data* is the data to be set in the node. *typename* is the string representation of the datatype, for example *int*, *string*, *compound*, \dots *lhid* is the `hid_t` reference to the datatype, is not necessary to specify unless a compound type is set.

NOTE: If the data to be set is of compound type, then the data should be of string type.

NOTE: If the node is a Reference node, the data should be set as a string type, where the data is the name of the referenced node.

setArrayValue(itemSize,dims,data,typename,lhid)

Sets an array value in the node instance. *itemSize* is used for specifying the size of the value in bytes. It is not necessary to specify unless a compound type is set. *dims* is a list of dimensions of the data. *data* is the data to be set in the node. *typename* is the string representation of the datatype, for example *int*, *string*, *compound*, \dots *lhid* is the `hid_t` reference to the datatype, is not necessary to specify unless a compound type should be set.

NOTE: If the data to be set is of compound type, the data should be of string type.

commit(datatype)

Marks a *TYPE_ID* node to be committed. *datatype* is the `hid_t` reference to the datatype.

name()

Returns the name of the node instance.

type()

Returns the type of the node instance.

dims()

Returns a list of the dimensions of the node instance

format()

Returns the string representation of the node's datatype.

data()

Returns the fixed data of the node instance.

NOTE: If the data is of compound type, the data will be returned as a string.

rawdata()

Returns the raw data of the node instance.

NOTE: If the data is of compound type, the data will be returned as a string.

compound_data()

Returns a dictionary with all attributes in the compound attribute, it only works if the node instance is a compound attribute.

8.4 Examples

The creation of HDF5 files with PyHL is quite easy, and there are not to many things one has to know about the HDF5 internals. However, in order to build an HDF5 file, one has to understand that the file should be built sequentially, i.e. it is not possible to create a subgroup to a group before the group has been created. Neither is it possible to create an attribute or a dataset in a group before the group has been created etc. In other words, always create the top nodes before trying to create nodes under them in the heirarchy. Another thing to bear in mind is that when the method *addNode* has been called the nodelist instance will take control over the node, so it will not be possible to alter the node after a call to *addNode* has been made. When working with compound types, remember that the data that is passed to *setScalarValue* and *setArrayValue* must be a Python string. Also when working with compound types, the *itemSize* and *lhid* has to be passed on, otherwise the compound data most likely will be corrupted. Also when working with compound types, be aware that the hdf5 library has to be linked dynamically, otherwise it will not be possible to pass the *hid_t* references between the Python modules. Time to look at some simple examples. Comments will be written in *italics* and the actual code will be written in **bold face**.

8.4.1 Writing a simple HDF5 file

```
import _pyhl
from Numeric import *
# Create an empty node list instance
aList = _pyhl.nodelist()
# Create an group called info
aNode = _pyhl.node(_pyhl.GROUP_ID, "/info")
# Add the node to the nodelist
# Remember that the nodelist takes responsibility
aList.addNode(aNode)
# Insert the attribute xscale in the group "/info"
aNode = _pyhl.node(_pyhl.ATTRIBUTE_ID, "/info/xscale")
# Set the value to a double with value 10.0
# Note the -1's that has been used since the data not is compaound
aNode.setScalarValue(-1, 10.0, "double", -1)
aList.addNode(aNode)
# Similar for yscale, xsize and ysize
aNode = _pyhl.node(_pyhl.ATTRIBUTE_ID, "/info/yscale")
aNode.setScalarValue(-1, 20.0, "double", -1)
aList.addNode(aNode)
aNode = _pyhl.node(_pyhl.ATTRIBUTE_ID, "/info/xsize")
aNode.setScalarValue(-1, 10, "int", -1)
aList.addNode(aNode)
aNode = _pyhl.node(_pyhl.ATTRIBUTE_ID, "/info/ysize")
aNode.setScalarValue(-1, 10, "int", -1)
aList.addNode(aNode)
# Add a description
aNode = _pyhl.node(_pyhl.ATTRIBUTE_ID, "/info/description")
```

```
aNode.setScalarValue(-1,"This is a simple example","string",-1)
aList.addNode(aNode)
# Add an array of data
myArray = arange(100)
myArray = array(myArray.astype('i'),'i')
myArray = reshape(myArray,(10,10))
aNode = _pyhl.node(_pyhl.DATASET_ID,"/data")
# Set the data as an array, note the list with [10,10] which
# Indicates that it is an array of 10x10 items
aNode.setArrayValue(-1,[10,10],myArray,"int",-1)
aList.addNode(aNode)
# And now just write the file as "simple_test.hdf" with
# Compression level 9 (highest compression)
aList.write("simple_test.hdf",9)
```

When checking this file with `h5dump`, the command syntax would be: `prompt% h5dump simple_test.hdf` And the result would be:

```
HDF5 "simple_test.hdf" {
GROUP "/" {
  DATASET "data" {
    DATATYPE { H5T_STD_I32LE }
    DATASPACE { SIMPLE ( 10, 10 ) / ( 10, 10 ) }
    DATA {
      0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
      10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
      20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
      30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
      40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
      50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
      60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
      70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
      80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
      90, 91, 92, 93, 94, 95, 96, 97, 98, 99
    }
  }
}
GROUP "info" {
  ATTRIBUTE "xscale" {
    DATATYPE { H5T_IEEE_F64LE }
    DATASPACE { SCALAR }
    DATA {
      10
    }
  }
  ATTRIBUTE "yscale" {
    DATATYPE { H5T_IEEE_F64LE }
    DATASPACE { SCALAR }
    DATA {
      20
    }
  }
  ATTRIBUTE "xsize" {
    DATATYPE { H5T_STD_I32LE }
    DATASPACE { SCALAR }
    DATA {
      10
    }
  }
}
```

```

ATTRIBUTE "ysize" {
  DATATYPE { H5T_STD_I32LE }
  DATASPACE { SCALAR }
  DATA {
    10
  }
}
ATTRIBUTE "description" {
  DATATYPE {
    { STRSIZE 25;
      STRPAD H5T_STR_NULLTERM;
      CSET H5T_CSET_ASCII;
      CTYPE H5T_C_S1;
    }
  }
  DATASPACE { SCALAR }
  DATA {
    "This is a simple example"
  }
}
}
}
}

```

8.4.2 Writing an HDF5 file containing a compound datatype

This is a bit more complex since it requires the implementation of a Python `\C{}`-module that contains the datatype definition, and a couple of methods for converting data to a string and the other way around. There is a small example located in the `hlhdf/pyhl` directory called `rave_info_type` which implements a small compound type definition. Basically this module defines an object containing `xscale`, `yscale`, `xsize` and `ysize` variables. This module has also got a type class which should be used.

```

import _pyhl
import _rave_info_type
# Create the rave info HDF5 type
typedef = _rave_info_type.type()
# Create the rave info HDF5 object
obj = _rave_info_type.object()
# Set the values
obj.xsize=10
obj.ysize=10
obj.xscale=150.0
obj.yscale=150.0
aList = _pyhl.nodelist()
# Create a datatype node
aNode = _pyhl.node(_pyhl.TYPE_ID,"/MyDatatype")
# Make the datatype named
aNode.commit(typedef.hid())
aList.addNode(aNode)
# Create an attribute containing the compound type
aNode = _pyhl.node(_pyhl.ATTRIBUTE_ID,"/myCompoundAttribute")
# Note that I use both itemSize and lhid
# Also note how I translate the compound object to a string
aNode.setScalarValue(typedef.size(),obj.tostring(),"compound",typedef.hid())
aList.addNode(aNode)
# Better create a dataset also with the compound type
obj.xsize=1
obj.ysize=1

```

```
aNode = _pyhl.node(_pyhl.DATASET_ID, "/myCompoundDataset")
# I use setArrayValue instead
aNode.setArrayValue(typedef.size(), [1], obj.tostring(), "compound", typedef.hid())
aList.addNode(aNode)
# And finally write the HDF5 file.
aList.write("compound_test.hdf")
```

When checking this file with `h5dump`, the command syntax would be: `prompt% h5dump compound_test.hdf` And the result would be:

```
HDF5 "compound_test.hdf" {
GROUP "/" {
  ATTRIBUTE "myCompoundAttribute" {
    DATATYPE {
      H5T_STD_I32LE "xsize";
      H5T_STD_I32LE "ysize";
      H5T_IEEE_F64LE "xscale";
      H5T_IEEE_F64LE "yscale";
    }
    DATASPACE { SCALAR }
    DATA {
      {
        [ 10 ],
        [ 10 ],
        [ 150 ],
        [ 150 ]
      }
    }
  }
  DATATYPE "MyDatatype" {
    H5T_STD_I32LE "xsize";
    H5T_STD_I32LE "ysize";
    H5T_IEEE_F64LE "xscale";
    H5T_IEEE_F64LE "yscale";
  }
  DATASET "myCompoundDataset" {
    DATATYPE {
      "/MyDatatype"
    }
    DATASPACE { SIMPLE ( 1 ) / ( 1 ) }
    DATA {
      {
        [ 1 ],
        [ 1 ],
        [ 150 ],
        [ 150 ]
      }
    }
  }
}
}
```

8.4.3 Reading a simple HDF5 file

The following example code will read the `/info/xscale`, `/info/yscale` and `/data` fields from the HDF5 file `simple_test.hdf`.

```
# Read the file
aList = _pyhl.read_nodelist("simple_test.hdf")
# Select individual nodes, instead of all of them
```

```

aList.selectNode("/info/xscale")
aList.selectNode("/info/yscale")
aList.selectNode("/data")
# Fetch the data for selected nodes
aList.fetch()
# Print the data
aNode = aList.getNode("/info/xscale")
print "XSCALE=" + 'aNode.data()'
aNode = aList.getNode("/info/yscale")
print "YSCALE=" + 'aNode.data()'
aNode = aList.getNode("/data")
print "DATA=" + 'aNode.data()'

```

8.4.4 Reading an HDF5 file containing a compound type

This example shows how an HDF5 file containing a compound type in it can be read. It will read the file "compound_test.hdf" that was generated above. Note that this code might not be portable to any other machine due to the usage of the rawdata method.

```

import _pyhl
import _rave_info_type
# There is no meaning creating the type
obj = _rave_info_type.object()
aList = _pyhl.read_nodelist("compound_test.hdf")
# Select everything for retrieval
aList.selectAll()
aList.fetch()
aNode = aList.getNode("/myCompoundAttribute")
# Translate from the string representation to object
obj.fromstring(aNode.rawdata())
# Display the values
print "XSIZE="+'obj.xsize'
print "YSIZE="+'obj.ysize'
print "XSCALE="+'obj.xscale'
print "YSCALE="+'obj.yscale'

```

8.4.5 Reading an HDF5 file containing a compound type (alternative)

This example shows how an HDF5 file containing a compound type in it can be read. It will read the file "compound_test.hdf" that was generated above. This example should work on any supported platform.

```

import _pyhl
import _rave_info_type
# There is no meaning creating the type
obj = _rave_info_type.object()
aList = _pyhl.read_nodelist("compound_test.hdf")
# Select everything for retrieval
aList.selectAll()
aList.fetch()
aNode = aList.getNode("/myCompoundAttribute")
# Translate from the string representation to object
cdescr = obj.compound_data()
obj.xsize = cdescr["xsize"]
obj.ysize = cdescr["ysize"]
obj.xscale = cdescr["xscale"]
obj.yscale = cdescr["yscale"]
# Display the values
print "XSIZE="+'obj.xsize'
print "YSIZE="+'obj.ysize'

```



```
print "XSCALE="+obj.xscale'  
print "YSCALE="+obj.yscale'
```

8.4.6 Creating a HDF5 image with a reference

This example shows how it is possible to create a HDF5 that is viewable in for example the H5View visualization tool.

```
import _pyhl  
from Numeric import *  
# Function for creating a dummy palette  
def createPalette():  
    a=zeros((256,3),'b')  
    for i in range(0,256):  
        a[i][0]=i  
    return a  
# Function for creating a dummy image}  
  
def createImage():  
    a=zeros((256,256),'b')  
    for i in range(0,256):  
        for j in range(0,256):  
            a[i][j] = i  
    return a  
# Function for the HDF5 file  
def create_test_image():  
    a=_pyhl.nodelist()  
    # First create the palette}  
  
    b=_pyhl.node(_pyhl.DATASET_ID,"/PALETTE")  
    c=createPalette()  
    b.setArrayValue(1,[256,3],c,"uchar",-1)  
    a.addNode(b)  
    b=_pyhl.node(_pyhl.ATTRIBUTE_ID,"/PALETTE/CLASS")  
    b.setScalarValue(-1,"PALETTE","string",-1)  
    a.addNode(b)  
    b=_pyhl.node(_pyhl.ATTRIBUTE_ID,"/PALETTE/PAL_VERSION")  
    b.setScalarValue(-1,"1.2","string",-1)  
    a.addNode(b)  
    b=_pyhl.node(_pyhl.ATTRIBUTE_ID,"/PALETTE/PAL_COLORMODEL")  
    b.setScalarValue(-1,"RGB","string",-1)  
    a.addNode(b)  
    b=_pyhl.node(_pyhl.ATTRIBUTE_ID,"/PALETTE/PAL_TYPE")  
    b.setScalarValue(-1,"STANDARD8","string",-1)  
    a.addNode(b)  
    # Now create the image to display}  
  
    b=_pyhl.node(_pyhl.DATASET_ID,"/IMAGE1")  
    c=createImage()  
    b.setArrayValue(1,[256,256],c,"uchar",-1)  
    a.addNode(b)  
    b=_pyhl.node(_pyhl.ATTRIBUTE_ID,"/IMAGE1/CLASS")  
    b.setScalarValue(-1,"IMAGE","string",-1)  
    a.addNode(b)  
    b=_pyhl.node(_pyhl.ATTRIBUTE_ID,"/IMAGE1/IMAGE_VERSION")  
    b.setScalarValue(-1,"1.2","string",-1)  
    a.addNode(b)  
    # Finally insert the reference}
```

```
b=_pyhl.node(_pyhl.REFERENCE_ID,"/IMAGE1/PALETTE")
b.setScalarValue(-1,"/PALETTE","string",-1)
a.addNode(b)
a.write("ahewrittenimage.hdf")
# The main function}
if __name__=="__main__"
    create_test_image()
```