RFC: Adding support for 16-bit floating point and Complex number datatypes to HDF5

Jordan Henderson

There has long been interest in having first-class support for half-precision floating-point and complex number datatypes in HDF5 to better facilitate applications that work with these types of data. This RFC discusses adding support for these datatypes, as well as a few other convenient datatypes, to the HDF5 library.

Contents

1. Introduction

HDF5 has a rich set of predefined datatypes to cover some of the most common use cases when storing data in HDF5 files. However, there are several types of data in wide use today that HDF5 does not handle in a convenient or efficient manner, causing users to have to workaround this lack of support in ways that are often cumbersome.

One of the goals that this RFC intends to accomplish is enabling convenient and efficient storage and retrieval of half-precision floating-point data in HDF5 files. An HDF5 user can store their data in a dataset using one of the existing (at least 32-bit) floating-point types and later convert the data to half-precision floating-point in memory for calculations, but as the reduced storage requirements for 16-bit floating-point values is one of the main benefits of the datatype, this isn't an ideal scenario. While one can already create an HDF5 datatype that will allow storing of floating-point data as 16-bit values on disk, the process of doing so with HDF5's `H5T` API routines may not be obvious to users, meaning that applications could potentially benefit from having a convenient predefined datatype to use. Further, datatype conversions on datatypes like this which HDF5 doesn't have native support for can have very poor performance, as demonstrated in issue #2154 in HDF5's GitHub repository. This is due to HDF5 performing conversions on these datatypes in software rather than allowing the compiler to potentially perform these conversions in hardware through dedicated instructions. Once again, an HDF5 user could work around this issue by defining their own datatype conversion routine, but it would be better and more convenient for HDF5 to be able to perform fast conversions by default where possible.

Another problematic topic in the world of HDF5 data involves the storage of complex numbers. Complex numbers are used throughout the scientific and other communities, but HDF5 does not currently have native support for them. This has led HDF5 users to invent their own conventions for storing this type of data, which can be inconvenient and can result in HDF5 files that have poor interoperability. A feature request for native complex number support in HDF5 was made fairly recently and contains a good discussion on the issues surrounding this problem.

To address these issues and work toward more portable HDF5 files, this RFC proposes adding native support in HDF5 for several datatypes that have been requested in the past, as well as datatypes that will be useful in future workflows which use HDF5.

2. New datatypes

Support for the following C datatypes is being considered for addition to the HDF5 library: `_Float16`, `__bf16`, `_Bool`, `float _Complex`, `double _Complex`, `long double _Complex`.

2.1. `_Float16`

This is a half-precision floating-point binary interchange format type specified in **ISO/IEC TS 18661-3:2015**. It represents a binary16-format floating-point type consisting of:

- 16 bits of storage
- 1 bit for the sign

- 5 bits for the exponent
- 10 bits for the fraction

An alternative to this type is the `__fp16` type, but arithmetic operations on this type promote to `float` or higher, while arithmetic can often be performed directly on values of the `_Float16` type. It is also generally recommended that portable code use the `_Float16` type, as it is standardized by the C standards committee. Due to this, only support for the `_Float16` type is being considered here; support for the `__fp16` type may be considered in the future.

2.2. `__bf16`

This is a half-precision floating-point type developed by Google Brain which is useful for machine learning applications and other workflows where the reduced storage requirements and potentially faster calculation speeds of the format are important. It represents a bfloat16-format floating-point type consisting of:

- 16 bits of storage
- 1 bit for the sign
- 8 bits for the exponent
- 7 bits for the fraction

This format has not been formally standardized by the C standards committee and support varies across platforms, so it may be tricky to support this format.

2.3. `_Bool`

This is the boolean datatype introduced in the C99 standard. The main reason for including native support for this type in HDF5 is to be able to reduce its storage requirements by not having to rely on integer types for storing boolean values.

2.4. `float _Complex`, `double _Complex`, `long double _Complex`

These are types representing complex numbers consisting of a real and imaginary part, each of which is a floating-point value of the specified type (`float`, `double` or `long double`). These types have been discussed more extensively in RFC: New Datatypes and are simply brought up once again for discussion around inclusion into HDF5.

2.5. Other lesser-precision floating-point datatypes

Consideration has also been given towards supporting lesser-precision floating-point datatypes, such as 8-bit floating-point datatypes. However, current compiler support for this type seems to be minimal or nonexistent across a variety of platform/compiler combinations. Therefore, support for these types would not be implemented at this time, but may be considered in the future if they are found to be useful.

3. Approach

3.1. Configure time checks

For each new datatype added, configure-time checks will be added to both CMake and the Autotools to detect the size of the native form of each datatype. New macros in the form of `H5_SIZEOF_type` will be added to the files that generate the `H5pubconf.h` header file. If a compiler doesn't support a particular type, the macro for that type will have a value of 0.

3.2. New datatype IDs and macros

For each new datatype added, variables will be added to `H5T.c` for the IDs of library-internal datatypes which will represent the platform-native form of each datatype, as well as the "standard" file format storage form of each datatype. Then, the following new macros will be added for the new datatypes:

3.2.1. _Float16

`H5T_NATIVE_FLOAT16`

Maps to the ID of the library-internal datatype representing the platform's native representation of an IEEE binary16-format 16-bit floating point number.

NOTE: If the platform doesn't support the `_Float16` type, this macro will map to `H5I_INVALID_HID` and will cause an "invalid datatype" error if used for any HDF5 operation which is passed a datatype identifier. In this case, one should use the `H5T_IEEE_F16LE` or `H5T_IEEE_F16BE` macro when writing from or reading into buffers of binary16-format data.

`H5T_IEEE_F16LE`

Maps to the ID of the library-internal datatype representing a binary16-format 16-bit little-endian IEEE floating-point number.

`H5T_IEEE_F16BE`

Maps to the ID of the library-internal datatype representing a binary16-format 16-bit big-endian IEEE floating-point number.

3.2.2. __bf16

`H5T_NATIVE_BFLOAT16`

Maps to the ID of the library-internal datatype representing the platform's native representation of a bfloat16-format 16-bit floating point number.

NOTE: If the platform doesn't support the `__bf16` type, this macro will map to `H5I_INVALID_HID` and will cause an "invalid datatype" error if used for any HDF5 operation which is passed a datatype identifier. In this case, one should use the `H5T_BRAIN_F16LE` or `H5T_BRAIN_F16BE` macro when writing from or reading into buffers of bfloat16-format data.

`H5T_BRAIN_F16LE`

Maps to the ID of the library-internal datatype representing a bfloat16-format 16-bit little-endian floating-point number.

`H5T_BRAIN_F16BE`

Maps to the ID of the library-internal datatype representing a bfloat16-format 16-bit big-endian floating-point number.

3.2.3. `_Bool`

`H5T_NATIVE_BOOL`

Maps to the ID of the library-internal datatype representing the platform's native representation of a `_Bool` boolean type.

`H5T_STD_BOOL`

Maps to the ID of the library-internal datatype representing a 1-bit boolean value.

3.2.4. `float _Complex`, `double _Complex`, `long double _Complex`

`H5T_NATIVE_FLOAT_COMPLEX`

Maps to the ID of the library-internal datatype representing the platform's native representation of a complex number consisting of `float` values.

NOTE: If the platform doesn't support the `float _Complex` type, this macro will map to `H5I_INVALID_HID` and will cause an "invalid datatype" error if used for any HDF5 operation which is passed a datatype identifier. In this case, one should use the `H5T_STD_FLOAT_COMPLEX_LE` or `H5T_STD_FLOAT_COMPLEX_BE` macro when writing from or reading into buffers of complex numbers consisting of 32-bit-precision floating-point values.

`H5T_STD_FLOAT_COMPLEX_LE`

Maps to the ID of the library-internal datatype representing a complex number consisting of 2 32-bit-precision little-endian floating-point values.

`H5T_STD_FLOAT_COMPLEX_BE`

Maps to the ID of the library-internal datatype representing a complex number consisting of 2 32-bit-precision big-endian floating-point values.

`H5T_NATIVE_DOUBLE_COMPLEX`

Maps to the ID of the library-internal datatype representing the platform's native representation of a complex number consisting of `double` values.

NOTE: If the platform doesn't support the `double _Complex` type, this macro will map to `H5I_INVALID_HID` and will cause an "invalid datatype" error if used for any HDF5 operation which is passed a datatype identifier. In this case, one should use the `H5T_STD_DOUBLE_COMPLEX_LE` or `H5T_STD_DOUBLE_COMPLEX_BE` macro when writing from or reading into buffers of complex numbers consisting of 64-bit-precision floating-point values.

`H5T_STD_DOUBLE_COMPLEX_LE`

Maps to the ID of the library-internal datatype representing a complex number consisting of 2 64-bit-precision little-endian floating-point values.

`H5T_STD_DOUBLE_COMPLEX_BE`

Maps to the ID of the library-internal datatype representing a complex number consisting of 2 64-bit-precision big-endian floating-point values.

`H5T_NATIVE_LDOUBLE_COMPLEX`

Maps to the ID of the library-internal datatype representing the platform's native representation of a complex number consisting of `long double` values.

NOTE: If the platform doesn't support the `long double _Complex` type, this macro will map to `H5I_INVALID_HID` and will cause an "invalid datatype" error if used for any HDF5 operation which is passed a datatype identifier. In this case, one should use the `H5T_STD_LDOUBLE_COMPLEX_LE` or `H5T_STD_LDOUBLE_COMPLEX_BE` macro when writing from or reading into buffers of complex numbers consisting of 128-bit-precision floating-point values.

`H5T_STD_LDOUBLE_COMPLEX_LE`

Maps to the ID of the library-internal datatype representing a complex number consisting of 2 128-bit-precision little-endian floating-point values.

`H5T_STD_LDOUBLE_COMPLEX_BE`

Maps to the ID of the library-internal datatype representing a complex number consisting of 2 128-bit-precision big-endian floating-point values.

3.3. On-disk format for new datatypes

3.3.1. `_Float16`

Data of this type will be stored as 16-bit values on disk, according to the endian-ness of the specified datatype (e.g. `H5T_IEEE_F16LE` vs. `H5T_IEEE_F16BE`), in a similar format to how other floating-point values are stored in HDF5.

3.3.2. `__bf16`

Data of this type will be stored as 16-bit values on disk, according to the endian-ness of the specified datatype (e.g. `H5T_BRAIN_F16LE` vs. `H5T_BRAIN_F16BE`), in a similar format to how other floating-point values are stored in HDF5.

3.3.3. `_Bool`

Data of this type will be stored as a 1-bit value on disk.

3.3.4. `float _Complex`, `double _Complex`, `long double _Complex`

Data of these types will be stored as atomic 64-bit, 128-bit, or 256-bit values on disk, respectively, according to the endian-ness of the specified datatype. The benefit to choosing this format is that these values should be able to be converted directly from disk format to a platform's in-memory representation with zero or near-zero overhead during conversion (depending on the endian-ness of the disk format and the in-memory representation). However, this means that there will likely be some overhead during conversion from disk format to in-memory representation for applications that use their own convention, such as a compound datatype with two members, for representing complex numbers.

NOTE: These on-disk formats have been chosen as they map to common sizes (multiplied by 2) of `float`, `double` and `long double` across platforms. Care should be taken when interacting with these datatypes on a platform where the size of the C types don't match these sizes.

3.4. Conversions

For each new datatype added, hard and soft conversion routines will be added to `H5Tconv.c` to convert between the new type and each existing native integer and floating-point type. In particular, the conversions for complex number types will follow the rules laid out in RFC: New Datatypes. Additionally, some conversions between complex number types and existing user-defined representations (e.g., a compound type with two members, an array type with two elements, etc.) will be added to help maintain compatibility with those representations.

3.5. `H5trace`

For each new datatype added, a printout for the new datatype will be added to the `H5_trace_args` function in `H5trace.c`.

3.6. Testing

For each new datatype added, tests will be added to the library to exercise both using the new datatype to create objects, as well as converting between the new datatype and existing HDF5 datatypes (see `test/dt_arith.c`).

4. HDF5 tools support

After adding these new datatypes, HDF5's tools library and tool applications will need to be updated to understand these new types. At minimum, this will involve updating the tools library's object dumping routines in `h5tools_dump.c`, the `print_type` routine in `h5diff_util.c` and the `print_native_type` routine in `h5ls.c`. Additionally, tools tests which exercise using the tool applications on these new types should be added to the library's existing tests.

As part of implementing support for these datatypes, it must be decided how the tool applications should print out a visual representation for complex datatypes when encountered. This RFC proposes the following as a starting point for discussion:

```
COMPLEX { real: VALUE, imag: VALUE }
```

5. HDF5 language wrappers support

The following sections briefly detail implementing support for these new datatypes in the HDF5 language wrappers both internal and external to HDF5.

5.1. Fortran

5.1.1. `_Float16`

Fortran support for this type appears to rely on compiler support; adding support in HDF5's Fortran wrappers may be straightforward or may take a bit of work, depending on compiler support across platforms.

5.1.2. `__bf16`

Fortran support for this type appears to rely on compiler support; adding support in HDF5's Fortran wrappers may be straightforward or may take a bit of work, depending on compiler support across platforms.

5.1.3. `_Bool`

Fortran has the `C_BOOL` type to support this datatype.

5.1.4. `float _Complex`, `double _Complex`, `long double _Complex`

Fortran has native support for this datatype; adding support in HDF5's Fortran wrappers should be straightforward.

5.2. Python

5.2.1. `_Float16`

`h5py` supports this datatype through NumPY.

5.2.2. `__bf16`

NumPY does not currently appear to support this type, but there has been some discussion around supporting it, at which point `h5py` could support it as well.

5.2.3. `_Bool`

`h5py` supports this datatype through NumPY.

5.2.4. `float _Complex`, `double _Complex`, `long double _Complex`

`h5py` supports this datatype through NumPY.

5.3. C++

5.3.1. `_Float16`

C++ has support for this type as an optional feature; adding support in HDF5's C++ wrappers may be straightforward or may take a bit of work, depending on compiler support across platforms.

5.3.2. `__bf16`

C++ has support for this type as an optional feature; adding support in HDF5's C++ wrappers may be straightforward or may take a bit of work, depending on compiler support across platforms.

5.3.3. `_Bool`

C++ has native support for this datatype; adding support in HDF5's C++ wrappers should be straightforward.

5.3.4. `float _Complex`, `double _Complex`, `long double _Complex`

C++ has native support for these datatypes; adding support in HDF5's C++ wrappers should be straightforward.

5.4. Java

5.4.1. `_Float16`

Java JDK 20 appears to introduce support for half-precision floating-point types, though they seem to represent these values in the `short` datatype. For a Java JDK before this version, support for this type could be implemented in HDF5's Java wrappers using one of several different approaches.

5.4.2. `__bf16`

Java doesn't support this datatype, but it could be implemented manually in HDF5's Java wrappers.

5.4.3. `_Bool`

Java has native support for this datatype; adding support in HDF5's Java wrappers should be straightforward.

5.4.4. `float _Complex`, `double _Complex`, `long double _Complex`

Support for complex numbers in HDF5's Java wrappers could be implemented through a Class.

6. Further considerations

The following items are miscellaneous considerations to take into account when interacting with these new datatypes:

- As there is no `printf` format specifier for the `_Float16` C type, and since a variable of type `_Float16` will not be promoted to a larger type when passed to `printf`, one should cast a variable of this type to `double` and use the `%f` specifier to print the value of such a variable.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Fusion Energy Sciences, under Award Number DE-SC0024442.

Revision History

Version Number	Date	Comments
v1	Jan. 29, 2024	Version 1 drafted.

A. Appendix: Portability

The following items must be taken into account for supporting these new datatypes across different platforms and compilers.

A.1. `_Float16`

A.1.1. GCC

This type is supported by GCC on:

- x86 systems where SSE2 support is available
- ARM systems when the `-mfp16-format=ieee` flag is used to specify that the IEEE half-precision floating-point format should be used
- AArch64 systems by default

A.1.2. Clang

Based on Clang Language Extensions, this type is supported by Clang on:

- x86 systems (natively if AVX512-FP16 is available; emulated otherwise)
- 32-bit ARM systems (emulated on some architectures; natively on some architectures)
- AArch64 systems (natively on ARMv8.2a and later)
- AMDGPU (natively)
- RISC-V (natively if Zfh or Zhinx support is available)

where ”natively” means the processor can directly perform arithmetic on the type using dedicated instructions and ”emulated” means the compiler will promote to `float` before performing arithmetic.

A.1.3. MSVC

As of the time of writing, Visual Studio 2022 does not appear to have support for any half-precision floating-point types.

A.1.4. NVHPC

NVHPC appears to support this type when compiling code used for CUDA compute (e.g., with `nvcc`), but does not appear to support this type when compiling regular C code with `nvc`.

A.1.5. OneAPI

Based on Intel oneAPI Math Function List, this type appears to be supported as of oneAPI version 2021.4 or higher with a ”next-generation Intel® Xeon® Scalable processor”. The Intel oneAPI Deep Neural Network

Library also appears to have support for a 16-bit floating-point type through the f16 type specifier, but it's not clear whether that type is more or less appropriate as an interchange format.

A.2. `__bf16`

A.2.1. GCC

This type is supported as of GCC 13 on:

- x86 systems where SSE2 support is available

A.2.2. Clang

Based on Clang Language Extensions, this type is supported by Clang on:

- x86 systems where SSE2 support is available
- 32-bit ARM systems
- AArch64 systems
- RISC-V systems

Note that support for the `__bf16` type is currently always emulated and not supported natively on any of the platforms above.

A.2.3. MSVC

As of the time of writing, Visual Studio 2022 does not appear to have support for any half-precision floating-point types.

A.2.4. NVHPC

This type appears to be supported by NVHPC as of version 23.9.

A.2.5. OneAPI

The Intel oneAPI Deep Neural Network Library (oneDNN) appears to have support for this type through the `bf16` type specifier. A whitepaper on the topic can be found at here.

A.3. `_Bool`

This type is supported on any C99-compliant compiler, which includes all the compilers mentioned for other types here.

A.4. `float _Complex`, `double _Complex`, `long double _Complex`

These types will generally be supported on any platform and compiler combination that implements the ISO C99 standard. However, it should be noted that support for complex numbers became optional in the C11 standard, meaning that C99 compliance is not enough of an indicator of support for these types. As of the C11 standard, there is a feature test macro, `__STDC_NO_COMPLEX__`, to determine this support, but as it only exists in the C11 standard and HDF5 only requires C99 compliance, additional configure time checks will likely be needed to check for complex number support. As of the C99 standard, there is also the feature test macro, `__STDC_IEC_559_COMPLEX__`, which determines whether the available complex number support conforms to IEC 60559. Notably, this involves supporting some additional macros and types for imaginary numbers. While in theory this macro could be used to check for complex number support, it is only a recommendation in the C99 standard that a compiler which defines this macro should support imaginary numbers. As of the C11 standard, it became mandatory for a compiler to support imaginary numbers if it defines this macro, but as of the C23 standard this macro has changed to `__STDC_IEC_60559_COMPLEX__`. Therefore, a configure time check for complex number support should likely rely on trying to compile a program that uses the complex number types rather than relying on these macros (though `__STDC_NO_COMPLEX__` can be checked if C11 support or later is available).

A.4.1. GCC

These types are fully supported, as long as the value of `__STDC_NO_COMPLEX__`, if available, is not `1`.

A.4.2. Clang

These types are fully supported, as long as the value of `__STDC_NO_COMPLEX__`, if available, is not `1`.

A.4.3. MSVC

Support for complex numbers is available, but MSVC does not expose the `float _Complex`, `double _Complex`, `long double _Complex` types and instead uses the `_Fcomplex`, `_Dcomplex`, `_Lcomplex` types, respectively.

A.4.4. NVHPC

These types are fully supported, as long as the value of `__STDC_NO_COMPLEX__`, if available, is not `1`.

A.4.5. OneAPI

As oneAPI is C99 compliant, these types should be fully supported, as long as the value of `__STDC_NO_COMPLEX__`, if available, is not `1`.

B. Appendix: Examples

The following are examples of what using these new datatypes should look like once support has been implemented in HDF5. Note that these examples are subject to change, depending on the final decisions made when implementing support for these new datatypes. Once support is implemented, relevant examples will be added to HDF5 to serve as a better guide.

Create a dataset with a binary16 little-endian 16-bit floating-point datatype

```
#include "hdf5.h"

int
main(int argc, char **argv)
{
    hsize_t dims[2] = { 10, 10 };
    hid_t file_id   = H5I_INVALID_HID;
    hid_t dset_id   = H5I_INVALID_HID;
    hid_t space_id  = H5I_INVALID_HID;

    file_id = H5Fcreate("file.h5", H5F_ACC_TRUNC,
                        H5P_DEFAULT, H5P_DEFAULT);

    space_id = H5Screate_simple(2, dims, NULL);

    dset_id = H5Dcreate2(file_id, "dset", H5T_IEEE_F16LE,
                         space_id, H5P_DEFAULT,
                         H5P_DEFAULT, H5P_DEFAULT);

    ...

    H5Sclose(space_id);
    H5Dclose(dset_id);
    H5Fclose(file_id);

    return 0;
}
```

Write to a binary16 little-endian 16-bit floating-point dataset using native `_Float16` type

```c
#include "hdf5.h"

int
main(int argc, char **argv)
{
    _Float16 data[100];
    hsize_t dims[2] = { 10, 10 };
    hid_t file_id   = H5I_INVALID_HID;
    hid_t dset_id   = H5I_INVALID_HID;
    hid_t space_id  = H5I_INVALID_HID;

    file_id = H5Fcreate("file.h5", H5F_ACC_TRUNC,
                        H5P_DEFAULT, H5P_DEFAULT);

    space_id = H5Screate_simple(2, dims, NULL);

    dset_id = H5Dcreate2(file_id, "dset", H5T_IEEE_F16LE,
                         space_id, H5P_DEFAULT,
                         H5P_DEFAULT, H5P_DEFAULT);

    for (size_t i = 0; i < 100; i++)
        data[i] = 1.1f16;

    H5Dwrite(dset_id, H5T_NATIVE_FLOAT16, H5S_ALL,
             H5S_ALL, H5P_DEFAULT, data);

    H5Sclose(space_id);
    H5Dclose(dset_id);
    H5Fclose(file_id);

    return 0;
}
```

Write to and read from a binary16 little-endian 16-bit floating-point dataset using native _Float16 type

```c
#include "hdf5.h"

int
main(int argc, char **argv)
{
    _Float16 data[100];
    _Float16 read_buf[100];
    hsize_t dims[2] = { 10, 10 };
    hid_t file_id   = H5I_INVALID_HID;
    hid_t dset_id   = H5I_INVALID_HID;
    hid_t space_id  = H5I_INVALID_HID;

    file_id = H5Fcreate("file.h5", H5F_ACC_TRUNC,
                        H5P_DEFAULT, H5P_DEFAULT);

    space_id = H5Screate_simple(2, dims, NULL);

    dset_id = H5Dcreate2(file_id, "dset", H5T_IEEE_F16LE,
                         space_id, H5P_DEFAULT,
                         H5P_DEFAULT, H5P_DEFAULT);

    for (size_t i = 0; i < 100; i++)
        data[i] = 1.1f16;

    H5Dwrite(dset_id, H5T_NATIVE_FLOAT16, H5S_ALL,
             H5S_ALL, H5P_DEFAULT, data);

    H5Dread(dset_id, H5T_NATIVE_FLOAT16, H5S_ALL,
            H5S_ALL, H5P_DEFAULT, read_buf);

    for (size_t i = 0; i < 100; i++)
        printf("%f\n", (double)read_buf[i]);

    H5Sclose(space_id);
    H5Dclose(dset_id);
    H5Fclose(file_id);

    return 0;
}
```

Write to and read from a little-endian 128-bit (2 64-bit doubles) complex number dataset using native `double _Complex` type

```
#include <complex.h>

#include "hdf5.h"

int
main(int argc, char **argv)
{
    double _Complex data[100];
    double _Complex read_buf[100];
    hsize_t dims[2] = { 10, 10 };
    hid_t file_id   = H5I_INVALID_HID;
    hid_t dset_id   = H5I_INVALID_HID;
    hid_t space_id  = H5I_INVALID_HID;

    file_id = H5Fcreate("file.h5", H5F_ACC_TRUNC,
                        H5P_DEFAULT, H5P_DEFAULT);

    space_id = H5Screate_simple(2, dims, NULL);

    dset_id = H5Dcreate2(file_id, "dset", H5T_STD_DOUBLE_COMPLEX_LE,
                         space_id, H5P_DEFAULT,
                         H5P_DEFAULT, H5P_DEFAULT);

    for (size_t i = 0; i < 100; i++)
        data[i] = 1.0 + 2.0*I;

    H5Dwrite(dset_id, H5T_NATIVE_DOUBLE_COMPLEX,
             H5S_ALL, H5S_ALL, H5P_DEFAULT, data);

    H5Dread(dset_id, H5T_NATIVE_DOUBLE_COMPLEX,
            H5S_ALL, H5S_ALL, H5P_DEFAULT, read_buf);

    for (size_t i = 0; i < 100; i++)
        printf("%f%+fi\n", creal(read_buf[i]), cimag(read_buf[i]));

    H5Sclose(space_id);
    H5Dclose(dset_id);
    H5Fclose(file_id);

    return 0;
}
```

Write to a little-endian 128-bit (2 64-bit doubles) complex number dataset when platform doesn't support `double _Complex` type

```
#include "hdf5.h"

int
main(int argc, char **argv)
{
    /* Note that data has twice the number of elements as the dataset */
    double data[200];
    hsize_t dims[2] = { 10, 10 };
    hid_t file_id   = H5I_INVALID_HID;
    hid_t dset_id   = H5I_INVALID_HID;
    hid_t space_id  = H5I_INVALID_HID;

    file_id = H5Fcreate("file.h5", H5F_ACC_TRUNC,
                        H5P_DEFAULT, H5P_DEFAULT);

    space_id = H5Screate_simple(2, dims, NULL);

    dset_id = H5Dcreate2(file_id, "dset", H5T_STD_DOUBLE_COMPLEX_LE,
                         space_id, H5P_DEFAULT,
                         H5P_DEFAULT, H5P_DEFAULT);

    for (size_t i = 0; i < 200; ) {
        data[i]     = 1.0; /* real part */
        data[i + 1] = 2.0; /* imaginary part */
        i += 2;
    }

    H5Dwrite(dset_id, H5T_STD_DOUBLE_COMPLEX_LE,
             H5S_ALL, H5S_ALL, H5P_DEFAULT, data);

    H5Sclose(space_id);
    H5Dclose(dset_id);
    H5Fclose(file_id);

    return 0;
}
```