Python IntelHex library Documentation

Release 2.1

Alexander Belchenko

May 16, 2017

Contents

Contents:

CHAPTER 1

Introduction

About

The Intel HEX file format is widely used in microprocessors and microcontrollers area as the de facto standard for code representation for microelectronic devices programming.

This work implements an **intelhex** Python library to read, write, create from scratch and manipulate data from HEX (also known as Intel HEX) file format. These operations are provided by `IntelHex` class.

The distribution package also includes several convenience Python scripts to do basic tasks that utilize this library. The `bin2hex.py` script converts binary data to HEX, and the `hex2bin.py` works the other direction. `hex2dump.py` converts data from HEX to a hexdump which is useful for inspecting data, and `hexmerge.py` merges multiple HEX files into one. In addition you can try inspecting differences between two HEX files with `hexdiff.py` utility which uses dump output similar to `hex2dump.py`.

You can find IntelHex library on PyPI:

https://pypi.python.org/pypi/IntelHex

on Launchpad:

https://launchpad.net/intelhex

on GitHub:

https://github.com/bialix/intelhex

Motivation

This work was partially inspired by SRecord software at the moment when I stuck with its limitations and unintuitive behavior.

So I've made this library and related tools which give me full control over data and HEX file creation. Not the best reason to start yet another project. But, as you probably know, nothing is better than scratch our own itches, especially if you want to re-implement something in your favorite programming language.

Over the years it turned out that my small python library was very useful to many people, and allowed them not only to create utilities to manipulate with HEX files, but also to create custom bootloaders for their devices.

I started writing this library in 2005, and now 10 years later it's still alive and useful to other developers. That keeps me working on improving the code, even though I don't work on embedding systems for some time.

If you find IntelHex library useful, please send me email and tell a bit about how you're using it and in which projects/areas. That will not only satisfy my curiosity but also will help me to keep working on this project.

License

The code distributed under the BSD license. See LICENSE.txt in sources archive.

Installation

Note: some commands below have *sudo* as first word. It's required only on Linux or Mac OS X. Omit the prefix if you're on Windows.

Installing with pip

If you just need IntelHex library installed as your system python library then it's better to use modern tool called `pip` (http://www.pip-installer.org/en/latest/) to install with the command:

```
sudo pip install intelhex
```

The latest versions of Python interpreter (like 2.7.9, or 3.4.x and later) have pip in the standard installer/distribution.

The simplest way to check whether you have pip installed is to check command (for Python 2.5+):

```
python -m pip list
```

If this does not work, you can install pip by downloading single file from this page: https://pip.pypa.io/en/latest/installing.html#install-pip and run it as

```
sudo python get-pip.py
```

Download sources

You can get archive with the latest released code, docs and other files from PyPI:

https://pypi.python.org/pypi/IntelHex

Also IntelHex may be downloaded from:

https://launchpad.net/intelhex/+download

You can get the archive with the unreleased code from GitHub page:

https://github.com/bialix/intelhex

Use the corresponding menu item in the right-hand side bar on that page (e.g. "Download ZIP").

Get source code with bzr or git

Get the latest development code with bzr:

bzr branch lp:intelhex

Or get the latest development code with git:

git clone https://github.com/bialix/intelhex.git

Install from sources

IntelHex has got stadard setup.py installation script. Assuming Python is properly installed on your platform, installation should require just running of the following command from the root directory of the sources:

```
sudo python setup.py install
```

This will install the intelhex package into your system's site-packages directory and place the helper scripts in your Python site-packages binaries directory. Once it is done, any other Python scripts or modules should be able to import the package using:

```
>>> from intelhex import IntelHex
```

The scripts should be in your PATH so that they could be called from anywhere in the file system.

See the Python distutils website for more information, or try typing, `python setup.py --help` from the root directory of the sources.

Note for Windows users

Please note that for historical reasons IntelHex library doesn't use setuptools for installation task, therefore we don't create exe-wrappers for helper scripts as hex2bin.py, bin2hex.py and other mentioned in this documentation (see section Convenience Scripts).

You can find these scripts in your python Script directory (usually *C:\PythonXY\Scripts*). You need either to create batch file to run them, or use Python interpreter:

python C:\PythonXY\Scripts\hex2bin.py ...

Python 3 compatibility

Intelhex library supports Python 2 (2.4-2.7) and Python 3 (3.2-3.5) without external libraries or 2to3 tool. Enjoy.

I've successfully run unit tests of IntelHex against following versions of Python:

2.4.4, 2.5.4, 2.6.6 (32/64 bits), 2.7.9 (32/64 bits), 3.2.5 (32/64 bits), 3.3.5 (32/64 bits), 3.4.3 (32/64 bits), 3.5.0a3 (32/64 bits), and also PyPy 2.5.1 (which is Python 2.7.9)

Which Python version should you use?

If you don't really know which version of Python (2 or 3) you should use with IntelHex then please check following pre-requisites:

1. Are you already have some Python installed on your computer and that version is supported by IntelHex (see above)?

2. Should you use another third-party libraries? If so, check their requirements.
3. Python 2.7 is the safest choice so far, but if you have a chance then try latest stable Python 3 version.

CHAPTER 2

Basic API and usage

Initializing the class

Example of typical initialization of `IntelHex` class:

```
>>> from intelhex import IntelHex
>>> ih = IntelHex("foo.hex")
```

In the second line we are creating an instance of the class. The constructor optionally takes data to initialize the class. This can be the name of the HEX file, a file-like object, a dictionary, or another instance of IntelHex. If specified, this source is automatically read and decoded. Because of the flexibility of file-like objects in Python, objects like sys.stdin can be used.

If the source is another instance of IntelHex, the new object will become a copy of the source. Finally, a Python dictionary may be specified. This dictionary should have keys equal to memory locations and values equal to the data stored in those locations. See the docstrings for more details.

Reading data

Once created, an IntelHex object can be loaded with data. This is only necessary if "source" was unspecified in the constructor. You can also load data several times (but if addresses in those files overlap you get exception `AddressOverlapError`). This error is only raised when reading from hex files. When reading from other formats, without explicitly calling `merge`, the data will be overwritten. E.g.:

```
>>> from intelhex import IntelHex
>>> ih = IntelHex()                      # create empty object
>>> ih.loadhex('foo.hex')                # load from hex
>>> ih.loadfile('bar.hex',format='hex') # also load from hex
>>> ih.fromfile('bar.hex',format='hex') # also load from hex
```

NOTE: using `IntelHex.fromfile` is recommended way.

All of the above examples will read from HEX files. IntelHex also supports reading straight binary files. For example:

```
>>> from intelhex import IntelHex
>>> ih = IntelHex()                      # create empty object
>>> ih.loadbin('foo.bin')                # load from bin
>>> ih.fromfile('bar.bin',format='bin') # also load from bin
>>> ih.loadbin('baz.bin',offset=0x1000) # load binary data and place them
>>>                                      # starting with specified offset
```

Finally, data can be loaded from an appropriate Python dictionary. This will permit you to store the data in an IntelHex object to a builtin dictionary and restore the object at a later time. For example:

```
>>> from intelhex import IntelHex
>>> ih = IntelHex('foo.hex') # create empty object
>>> pydict = ih.todict()     # dump contents to pydict
```

...do something with the dictionary...

```
>>> newIH = IntelHex(pydict) # recreate object with dict
>>> another = IntelHex()     # make a blank instance
>>> another.fromdict(pydict) # now another is the same as newIH
```

Basic data inspection

You can get or modify some data by address in the usual way: via Python indexing operations:

```
>>> print ih[0] # read data from address 0
```

When you need to work with 16-bit data stored in 8-bit Intel HEX files you need to use class `IntelHex16bit`. This class is derived from IntelHex and has all its methods. Some of methods have been modified to implement 16-bit behaviour.

NOTE: `IntelHex16bit` class despite its name **can't handle** real HEX16 files. Initially `IntelHex16bit` has been created as helper class to work with HEX files for Microchip's PIC16 family firmware. It may or may not work for your purpose.

This class assumes the data is in Little Endian byte order. The data can be accessed exactly like above, except that data returned will be 16 bits, and the addresses should be word addresses.

Another useful inspection tool is the dump command. This will output the entire contents of the hex file to stdout or to a specified file object like so:

```
>>> ih.dump() # dump contents of ih to stdout in tabular hexdump format

>>> f = open('hexdump.txt', 'w') # open file for writing
>>> ih.dump(f)                   # dump to file object
>>> f.close()                    # close file
```

More data inspection

IntelHex provides some metadata about the hex file it contains. To obtain address limits use methods `.minaddr()` and `.maxaddr()`. These are computed based on the lowest and highest used memory spaces respectively.

Some linkers write to produced HEX file information about start address (either record 03 or 05). IntelHex is able correctly read such records and store information internally in `start_addr` attribute that itself is either `None` or a dictionary with the address value(s).

When input HEX file contains record type 03 (Start Segment Address Record), `start_addr` takes value:

```
{'CS': XXX, 'IP': YYY}
```

Here:

- `XXX` is value of CS register
- `YYY` is value of IP register

To obtain or change `CS` or `IP` value you need to use their names as keys for `start_addr` dictionary:

```
>>> ih = IntelHex('file_with_03.hex')
>>> print ih.start_addr['CS']
>>> print ih.start_addr['IP']
```

When input HEX file contains record type 05 (Start Linear Address Record), `start_addr` takes value:

```
{'EIP': ZZZ}
```

Here `ZZZ` is value of EIP register.

Example:

```
>>> ih = IntelHex('file_with_05.hex')
>>> print ih.start_addr['EIP']
```

You can manually set required start address:

```
>>> ih.start_addr = {'CS': 0x1234, 'IP': 0x5678}
>>> ih.start_addr = {'EIP': 0x12345678}
```

To delete start address info give value `None` or empty dictionary:

```
>>> ih.start_addr = None
>>> ih.start_addr = {}
```

When you write data to HEX file you can disable writing start address with additional argument `write_start_addr`:

```
>>> ih.write_hex_file('out.hex') # by default writing start address
>>> ih.write_hex_file('out.hex', True) # as above
>>> ih.write_hex_file('out.hex', False) # don't write start address
```

When `start_addr` is `None` or an empty dictionary nothing will be written regardless of `write_start_addr` argument value.

For more information about start address, please see the Intel Hex file format specification.

Because Intel Hex files do not specify every location in memory, it is necessary to have a padding byte defined. Whenever a read is attempted from an address that is unspecified, the padding byte is returned. This default data is set via attribute `.padding` of class instance. This defaults to '0xFF', but it can be changed by the user like so:

```
>>> print ih[0]       # prints 0xFF because this location is blank
>>> ih.padding = 0x00 # change padding byte
>>> print ih[0]       # prints 0x00 because this location is blank
```

Summarizing the data chunks

One of the more useful properties of HEX files is that they can specify data in discontinuous segments. There are two main methods to summarize which data addresses are occupied:

```
>>> ih.addresses()
>>> ih.segments()
```

The first will return a list of occupied data addresses in sorted order. The second will return a list of 2-tuples objects, in sorted order, representing start and stop addresses of contiguous segment chunks of occupied data. Those 2-tuples are suitable to be used as `start` and `stop` arguments of standard `range` function.

Writing out data

Data contained in IntelHex can be written out in a few different formats, including HEX, bin, or python dictionaries.

You can write out HEX data contained in object by method `.write_hex_file(f)`. Parameter `f` should be filename or file-like object. Note that this can include builtins like sys.stdout. Also you can use the universal tofile.

To convert data of IntelHex object to HEX8 file format without actually saving it to disk you can use the builtin StringIO file-like object, e.g.:

```
>>> from cStringIO import StringIO
>>> from intelhex import IntelHex
>>> ih = IntelHex()
>>> ih[0] = 0x55
>>> sio = StringIO()
>>> ih.write_hex_file(sio)
>>> hexstr = sio.getvalue()
>>> sio.close()
```

Variable `hexstr` will contain a string with the content of a HEX8 file.

To write data as a hex file you also can use universal method `tofile`:

```
>>> ih.tofile(sio, format='hex')
```

NOTE: using IntelHex.tofile is recommended way.

Class IntelHex has several methods for converting data of IntelHex objects into binary form:

- `tobinarray` (returns array of unsigned char bytes);
- `tobinstr` (returns string of bytes);
- `tobinfile` (convert content to binary form and write to file).

Example:

```
>>> from intelhex import IntelHex
>>> ih = IntelHex("foo.hex")
>>> ih.tobinfile("foo.bin")
```

Also you can use universal method `tofile` to write data as binary file:

```
>>> ih.tofile("foo.bin", format='bin')
```

Writing data in chunks

If you need to get binary data from IntelHex as series of chunks then you can pass to `tobinarray`/`tobinstr` methods either start/end addresses or start address and required size of the chunk. This could be useful if you're creating Eeprom/Flash IC programmer or bootloader.

```
EEPROM_SIZE = 8192  # 8K bytes
BLOCK_SIZE = 128    # 128 bytes
for addr in range(0, EEPROM_SIZE, BLOCK_SIZE):
    eeprom.i2c_write(addr, ih.tobinarray(start=addr, size=BLOCK_SIZE))
```

Merging two hex files

IntelHex supports merging two different hex files into one. This is done by initializing one IntelHex object with data and calling its merge method:

```
>>> original = IntelHex("foo.hex")
>>> new = IntelHex("bar.hex")
>>> original.merge(new, overlap='replace')
```

Now original will contain foo.hex merged with bar.hex. The overlap parameter specifies what should be done when memory locations in the original object overlap with locations in the new object. It can take three options:

- `error` - stop and raise an exception (default)
- `ignore` - keep data from the original that contains data at overlapped address
- `replace` - use data from the new object that contains data at overlapped address

You can merge only part of other hex file by using slice index notation:

```
>>> original = IntelHex("foo.hex")
>>> new = IntelHex("bar.hex")
>>> original.merge(new[0x0F:0x3F])
```

Creating Intel Hex files from scratch

Some facilities are provided for synthesizing Intel Hex files from scratch. These can also be used to modify a hex file in place. Just as you can use indexed reads to retrieve data, you can use indexed writes to modify the file, e.g.:

```
>>> ih[1] = 0x55 # modify data at address 1
```

A common usage would be to read a hex file with IntelHex, use the above syntax to modify it, and then write out the modified file. The above command can be used on an empty IntelHex object to synthesize a hex file from scratch.

Another important feature helps work with C strings via `putsz`/`getsz`, e.g.:

```
>>> ih.putsz(0x100,"A string")
```

This places "A string" followed by a terminating NULL character in address 0x100. The `getsz` method similarly retrieves a null terminated string from a specified address like so:

```
>>> ih.getsz(0x100)
```

This should retrieve the "A string" we stored earlier.

Additionally, `puts`/`gets` can be used to retrieve strings of specific length from the hex file like so:

```
>>> ih.puts(0x100,"data")
>>> ih.gets(0x100,4)
```

The second command should retrieve the characters 'd','a','t','a'. These methods do not use terminating NULLs, so the data need not be interpreted as a string. One usage of these commands comes from the Python `struct` module. This module allows the programmer to specify a C struct, and it will allow conversion between the variables and a packed string representation for use with `puts`/`gets`. For example, suppose we need to deal with a struct containing a char, a short, and a float:

```
>>> import struct
>>> formatstring = 'chf' # see Python docs for full list of valid struct formats
>>> ih.puts(0x10, struct.pack(formatstring,'a',24,18.6)) # put data in hex file
>>> (mychar,myshort,myfloat) = struct.unpack(formatstring, ih.gets(0x10,7)
```

Now `mychar`, `myshort`, and `myfloat` should contain the original data (assuming `sizeof(float) = 4` on this platform, otherwise the size may be wrong).

Handling errors

Many of the methods in IntelHex throw Python exceptions during error conditions. These can be caught and handled using try...except blocks like so:

```
>>> try:
...     mystring = ih.gets(0x20,20)
>>> except intelhex.NotEnoughDataError:
...     print "There is not enough data at that location"
```

See the API docs for information about errors raised by IntelHex. They are all subclasses of IntelHexError, so the except block above could be used to catch all of them. If your application has a way to gracefully handle these exceptions, they should be caught. Otherwise, Python will exit with a descriptive error message about the uncaught exception.

See Appendix A for error classes hierarchy.

CHAPTER 3

Convenience Scripts

When IntelHex is installed and added to the system path, some scripts are available for usage. Each one is meant to be operated from the command line. They provide help if called incorrectly.

Script `hex2bin.py`

You can use `hex2bin.py` as handy hex-to-bin converter. This script is just a frontend for function `hex2bin` from intelhex package.

```
Usage:
    python hex2bin.py [options] INFILE [OUTFILE]

Arguments:
    INFILE      name of hex file for processing.
    OUTFILE     name of output file. If omitted then output
                will be writing to stdout.


Options:
    -h, --help              this help message.
    -p, --pad=FF            pad byte for empty spaces (hex value).
    -r, --range=START:END   specify address range for writing output
                            (hex value).
                            Range can be in form 'START:' or ':END'.
    -l, --length=NNNN,
    -s, --size=NNNN         size of output (decimal value).
```

Per example, converting content of foo.hex to foo.bin addresses from 0 to FF:

```
$ python hex2bin.py -r 0000:00FF foo.hex
```

Or (equivalent):

```
$ python hex2bin.py -r 0000: -s 256 foo.hex
```

Script `bin2hex.py`

You can use bin2hex.py as simple bin-to-hex convertor. This script is just a frontend for function `bin2hex` from intelhex package.

```
Usage:
    python bin2hex.py [options] INFILE [OUTFILE]

Arguments:
    INFILE      name of bin file for processing.
                Use '-' for reading from stdin.

    OUTFILE     name of output file. If omitted then output
                will be writing to stdout.

Options:
    -h, --help              this help message.
    --offset=N              offset for loading bin file (default: 0).
```

Script `hex2dump.py`

This is a script to dump a hex file to a hexdump format. It is a frontend for `dump` function in `IntelHex` class.

```
Usage:
    python hex2dump.py [options] HEXFILE

Options:
    -h, --help              this help message.
    -r, --range=START:END   specify address range for dumping
                            (ascii hex value).
                            Range can be in form 'START:' or ':END'.

Arguments:
    HEXFILE     name of hex file for processing (use '-' to read
                from stdin)
```

Script `hexmerge.py`

This is a script to merge two different hex files. It is a frontend for the `merge` function in `IntelHex` class.

```
Usage:
    python hexmerge.py [options] FILES...

Options:
    -h, --help              this help message.
    -o, --output=FILENAME   output file name (emit output to stdout
                            if option is not specified)
    -r, --range=START:END   specify address range for output
```

```
                            (ascii hex value).
                            Range can be in form 'START:' or ':END'.
    --no-start-addr         Don't write start addr to output file.
    --overlap=METHOD        What to do when data in files overlapped.
                            Supported variants:
                            * error -- stop and show error message (default)
                            * ignore -- keep data from first file that
                                        contains data at overlapped address
                            * replace -- use data from last file that
                                         contains data at overlapped address

Arguments:
    FILES       list of hex files for merging
                (use '-' to read content from stdin)

You can specify address range for each file in the form:

    filename:START:END

See description of range option above.

You can omit START or END, so supported variants are:

    filename:START:     read filename and use data starting from START addr
    filename::END       read filename and use data till END addr

Use entire file content:

    filename
or
    filename::
```

Script `hexdiff.py`

This is a script to diff context of two hex files.

To create human-readable diff this utility converts both hex files to hex dumps first, and then utility compares those hex dumps and produces unified diff output for changes.

```
hexdiff: diff dumps of 2 hex files.

Usage:
    python hexdiff.py [options] FILE1 FILE2

Options:
    -h, --help              this help message.
    -v, --version           version info.
```

Script `hexinfo.py`

This is a script to summarize a hex file's contents.

This utility creates a YAML-formatted, human-readable summary of a set of HEX files. It includes the file name, execution start address (if any), and the address ranges covered by the data (if any).

```
hexinfo: summarize a hex file's contents.

Usage:
    python hexinfo.py [options] FILE [ FILE ... ]

Options:
    -h, --help              this help message.
    -v, --version           version info.
```

CHAPTER 4

Embedding into other projects

IntelHex should be easy to embed in other projects. The directory `intelhex` containing `__init__.py` can be directly placed in a depending project and used directly. From that project the same import statements described above can be used to make the library work. From other projects the import statement would change to:

```
>>> from myproject.intelhex import IntelHex
```

Alternatively, the IntelHex package can be installed into the site-packages directory and used as a system package.

In either case, IntelHex is distributed with a BSD-style license. This permits you to use it in any way you see fit, provided that the package is appropriately credited.

If you're using IntelHex library in your open-source project, or your company created freely available set of tools, utilities or sdk based on IntelHex library - please, send me email (to alexander belchenko at gmail com) and tell something about your project. I'd like to add name of your project/company to page "Who Uses IntelHex".

CHAPTER 5

Appendix A. IntelHex Errors Hierarchy

- `IntelHexError` - base error
  - `HexReaderError` - general hex reader error
    - `AddressOverlapError` - data for the same address overlap
    - `HexRecordError` - hex record decoder base error
      - `RecordLengthError` - record has invalid length
      - `RecordTypeError` - record has invalid type (RECTYP)
      - `RecordChecksumError` - record checksum mismatch
      - `EOFRecordError` - invalid EOF record (type 01)
      - `ExtendedAddressRecordError` - extended address record base error
      - `ExtendedSegmentAddressRecordError` - invalid extended segment address record (type 02)
      - `ExtendedLinearAddressRecordError` - invalid extended linear address record (type 04)
      - `StartAddressRecordError` - start address record base error
      - `StartSegmentAddressRecordError` - invalid start segment address record (type 03)
      - `StartLinearAddressRecordError` - invalid start linear address record (type 05)
      - `DuplicateStartAddressRecordError` - start address record appears twice
      - `InvalidStartAddressValueError` - invalid value of start addr record
  - `BadAccess16bit` - not enough data to read 16 bit value
  - `NotEnoughDataError` - not enough data to read N contiguous bytes
  - `EmptyIntelHexError` - requested operation cannot be performed with empty object

CHAPTER 6

Indices and tables

- genindex
- modindex
- search