## You

In the python packages to be built from source, can you give me how I can define my setup.cfg if setup.py file is like this:

#from distutils.core import setup, Extension
from setuptools import setup, Extension

setup(name='rdrand',
version='1.5.0',
description="Python interface to Intel hardware rng",
long\_description= "".join(open('rdrandom.rst').readlines()),
author="Chris Stillson",
author\_email="[stillson@gmail.com](mailto:stillson@gmail.com)",
url='<https://github.com/stillson/rdrand>',
license="New BSD license",
ext\_modules=[Extension('\_rdrand', ['rdrand.c'])],
py\_modules = ['rdrand'],
keywords = ['rdrand', 'rdseed', "intel","hardware","random","number","generator","rng"],
data\_files=[('share', ['rdrandom.rst'])],
classifiers = ["Development Status :: 4 - Beta",
"Intended Audience :: Developers",
"Intended Audience :: Science/Research",
"License :: OSI Approved :: BSD License",
"Programming Language :: Python",
"Programming Language :: Python :: 2",
"Programming Language :: Python :: 3",
"Topic :: Security",
"Topic :: Security :: Cryptography",],
extras\_require={
'tests': [],
'docs': [
'sphinx >= 1.4',
'sphinx\_rtd\_theme']}
)

## Copilot

Sure, I can help with that. The setup.cfg file is used to define the metadata and options that are applied when running setup.py. Here's an example of how you might define a setup.cfg file based on your setup.py:

[metadata]

name = rdrand

version = 1.5.0

description = Python interface to Intel hardware rng

long\_description = file: rdrandom.rst

author = Chris Stillson

author\_email = stillson@gmail.com

url = https://github.com/stillson/rdrand

license = New BSD license

keywords = rdrand, rdseed, intel, hardware, random, number, generator, rng

classifiers =

Development Status :: 4 - Beta

Intended Audience :: Developers

Intended Audience :: Science/Research

License :: OSI Approved :: BSD License

Programming Language :: Python

Programming Language :: Python :: 2

Programming Language :: Python :: 3

Topic :: Security

Topic :: Security :: Cryptography

[options]

py\_modules = rdrand

install\_requires =

sphinx >= 1.4

sphinx\_rtd\_theme

[options.extras\_require]

tests =

docs =

sphinx >= 1.4

sphinx\_rtd\_theme

[options.data\_files]

share = rdrandom.rst

[build\_ext]

sources = rdrand.c

Please note that this is a basic example and you might need to adjust it according to your specific needs. For instance, you might need to specify dependencies in the install\_requires section under [options] if your package depends on other packages. Also, the sources under [build\_ext] is used to specify the source files for the extension module. If you have multiple source files, you can list them separated by a comma. For example, sources = file1.c, file2.c, file3.c.

Remember to replace the placeholders with your actual values. Hope this helps! Let me know if you have any other questions. 😊

## You

I want to build using mingw64 gcc. Can you include that option in above file?

## Copilot

Sure, you can specify the compiler you want to use in the setup.cfg file. If you want to use the mingw64 gcc compiler, you can add a [build] section and specify the compiler option. Here's how you can modify the setup.cfg:

[metadata]

name = rdrand

version = 1.5.0

description = Python interface to Intel hardware rng

long\_description = file: rdrandom.rst

author = Chris Stillson

author\_email = stillson@gmail.com

url = https://github.com/stillson/rdrand

license = New BSD license

keywords = rdrand, rdseed, intel, hardware, random, number, generator, rng

classifiers =

Development Status :: 4 - Beta

Intended Audience :: Developers

Intended Audience :: Science/Research

License :: OSI Approved :: BSD License

Programming Language :: Python

Programming Language :: Python :: 2

Programming Language :: Python :: 3

Topic :: Security

Topic :: Security :: Cryptography

[options]

py\_modules = rdrand

install\_requires =

sphinx >= 1.4

sphinx\_rtd\_theme

[options.extras\_require]

tests =

docs =

sphinx >= 1.4

sphinx\_rtd\_theme

[options.data\_files]

share = rdrandom.rst

[build\_ext]

sources = rdrand.c

[build]

compiler = mingw32

In the [build] section, the compiler option is set to mingw32. This tells distutils to use the mingw32 compiler, which is the MinGW compiler for Windows. Please note that you need to have the MinGW compiler installed on your system and it should be available in your system's PATH.

Remember to replace the placeholders with your actual values. Hope this helps! Let me know if you have any other questions. 😊

## You

Why not mingw64 instead of mongw32 in compiler section?

## Copilot

Great question! The mingw32 in the compiler option is a bit of a misnomer. It doesn't mean that it's only for 32-bit systems. The mingw32 compiler can actually produce both 32-bit and 64-bit code. The name mingw32 is used for historical reasons.

When you specify mingw32 as the compiler, Python's build system will use the MinGW compiler (gcc) available in your system's PATH. If you have the MinGW-w64 version of gcc installed (which supports both 32-bit and 64-bit), and it's available in your PATH, then it will be used to compile the extension.

So, even though it says mingw32, it can still use the mingw64 gcc compiler if that's what is installed on your system. I hope this clears up the confusion! Let me know if you have any other questions. 😊

## You

Some error in above. See the output below:

$ pip install .
Processing d:\programs\msys2\home\juhi123\downloads\rdrand
Preparing metadata (setup.py) ... error
error: subprocess-exited-with-error

× python setup.py egg\_info did not run successfully.
│ exit code: 1
╰─> [8 lines of output]
running egg\_info
creating D:\Programs\Msys2\tmp\pip-pip-egg-info-7rk2ox5f\rdrand.egg-info
writing D:\Programs\Msys2\tmp\pip-pip-egg-info-7rk2ox5f\rdrand.egg-info\PKG-INFO
writing dependency\_links to D:\Programs\Msys2\tmp\pip-pip-egg-info-7rk2ox5f\rdrand.egg-info\dependency\_links.txt
writing requirements to D:\Programs\Msys2\tmp\pip-pip-egg-info-7rk2ox5f\rdrand.egg-info\requires.txt
writing top-level names to D:\Programs\Msys2\tmp\pip-pip-egg-info-7rk2ox5f\rdrand.egg-info\top\_level.txt
writing manifest file 'D:\Programs\Msys2\tmp\pip-pip-egg-info-7rk2ox5f\rdrand.egg-info\SOURCES.txt'
error: error in setup.cfg: command 'build\_ext' has no such option 'sources'
[end of output]

## Copilot

I apologize for the confusion. The build\_ext command in setup.cfg does not have a sources option. The source files for the extension should be specified in the setup.py file itself, as you have done.

The setup.cfg file is mainly used to specify package metadata and options for various commands, but the actual building of extensions is handled by setup.py.

Here's the corrected setup.cfg:

[metadata]

name = rdrand

version = 1.5.0

description = Python interface to Intel hardware rng

long\_description = file: rdrandom.rst

author = Chris Stillson

author\_email = stillson@gmail.com

url = https://github.com/stillson/rdrand

license = New BSD license

keywords = rdrand, rdseed, intel, hardware, random, number, generator, rng

classifiers =

Development Status :: 4 - Beta

Intended Audience :: Developers

Intended Audience :: Science/Research

License :: OSI Approved :: BSD License

Programming Language :: Python

Programming Language :: Python :: 2

Programming Language :: Python :: 3

Topic :: Security

Topic :: Security :: Cryptography

[options]

py\_modules = rdrand

install\_requires =

sphinx >= 1.4

sphinx\_rtd\_theme

[options.extras\_require]

tests =

docs =

sphinx >= 1.4

sphinx\_rtd\_theme

[options.data\_files]

share = rdrandom.rst

[build]

compiler = mingw32

I hope this helps! Let me know if you have any other questions. 😊

## You

See specific error messages below. I think some issue in source C file:

Building wheels for collected packages: rdrand
Building wheel for rdrand (setup.py) ... error
error: subprocess-exited-with-error

× python setup.py bdist\_wheel did not run successfully.
│ exit code: 1
╰─> [17 lines of output]
running bdist\_wheel
running build
running build\_py
creating build
creating build\lib.win-amd64-cpython-312
copying rdrand.py -> build\lib.win-amd64-cpython-312
running build\_ext
building '\_rdrand' extension
creating build\temp.win-amd64-cpython-312
creating build\temp.win-amd64-cpython-312\Release
gcc -mdll -O -Wall -ID:\Programs\Msys2\home\juhi123.venv\include -ID:\Programs\Python\include -ID:\Programs\Python\Include -c rdrand.c -o build\temp.win-amd64-cpython-312\Release\rdrand.o
rdrand.c: In function 'get\_bits':
rdrand.c:378:9: error: implicit declaration of function 'bcopy' [-Wimplicit-function-declaration]
378 | bcopy((char\*)&rando, &data[num\_quads \* 8], num\_chars);
| ^~~~~
rdrand.c:378:9: warning: incompatible implicit declaration of built-in function 'bcopy' [-Wbuiltin-declaration-mismatch]
error: command 'D:\Programs\Msys2\ucrt64\bin\gcc.exe' failed with exit code 1
[end of output]

## Copilot

The error message indicates that the bcopy function is not recognized. This is likely because bcopy is not a standard C function and may not be available in all environments. It's a function from the BSD library, and it's not part of the standard C library.

In many environments, including Windows and some versions of Linux, bcopy is not available. Instead, the standard function memcpy is typically used to copy memory areas. The memcpy function is part of the standard C library and should be available in all environments where C is supported.

You can replace the bcopy function with memcpy in your rdrand.c file. Here's how you can do it:

memcpy(&data[num\_quads \* 8], (char\*)&rando, num\_chars);

Please note that the order of the arguments to memcpy is different from bcopy. The first argument is the destination, and the second argument is the source.

After making this change, try building your package again. If you encounter any more issues, feel free to ask! 😊

## You

How can I comment line containing bcopy - It is a C source file.

## Copilot

In C, you can comment out a line by adding // at the beginning of the line. Here's how you can comment out the line containing bcopy:

// bcopy((char\*)&rando, &data[num\_quads \* 8], num\_chars);

This will prevent the line from being executed when the code is compiled and run. If you want to add a multi-line comment, you can use /\* to start the comment and \*/ to end the comment. For example:

/\*

bcopy((char\*)&rando, &data[num\_quads \* 8], num\_chars);

Another line of code here

\*/

This will comment out both lines of code. I hope this helps! Let me know if you have any other questions. 😊

## You

See next errors after above has been resolved:

error: subprocess-exited-with-error

× python setup.py bdist\_wheel did not run successfully.
│ exit code: 1
╰─> [15 lines of output]
running bdist\_wheel
running build
running build\_py
creating build
creating build\lib.win-amd64-cpython-312
copying rdrand.py -> build\lib.win-amd64-cpython-312
running build\_ext
building '\_rdrand' extension
creating build\temp.win-amd64-cpython-312
creating build\temp.win-amd64-cpython-312\Release
gcc -mdll -O -Wall -ID:\Programs\Msys2\home\juhi123.venv\include -ID:\Programs\Python\include -ID:\Programs\Python\Include -c rdrand.c -o build\temp.win-amd64-cpython-312\Release\rdrand.o
D:\Programs\Msys2\tmp\cc8tGl91.s: Assembler messages:
D:\Programs\Msys2\tmp\cc8tGl91.s:377: Error: invalid instruction suffix for push'
D:\Programs\Msys2\tmp\cc8tGl91.s:380: Error: invalid instruction suffix for pop'
error: command 'D:\Programs\Msys2\ucrt64\bin\gcc.exe' failed with exit code 1
[end of output]

## Copilot

The error messages you're seeing are coming from the assembler, which is the part of the compiler that translates the C code into machine code. The push and pop instructions are used to put values on the stack and remove them, respectively.

The error messages indicate that the assembler doesn't recognize the suffix for the push and pop instructions. This is likely because the assembler is expecting 32-bit instructions (suffix l for pushl and popl), but the code is using 64-bit instructions (suffix q for pushq and popq).

This issue can occur if you're trying to compile 64-bit code with a compiler or assembler that's set up for 32-bit code. You mentioned that you're using mingw64 gcc, which should be capable of compiling 64-bit code. However, it's possible that the build environment is configured for 32-bit code.

Here are a few things you can try:

1. **Check your build environment**: Make sure your build environment is set up for 64-bit code. If you're using a build system or IDE, check the settings to make sure it's configured for 64-bit builds.
2. **Check the compiler flags**: The -m64 flag tells gcc to generate 64-bit code. If this flag is not present, gcc might be generating 32-bit code. You can add this flag to the gcc command in your setup.py file.
3. **Check the source code**: If the source code contains inline assembly, make sure the assembly code is compatible with 64-bit processors. You might need to modify the assembly code or use a different function that's compatible with 64-bit processors.

I hope this helps! Let me know if you have any other questions. 😊

## You

See my C source. Where Can I make changes?

#define PY\_SSIZE\_T\_CLEAN
#include <Python.h>
#include <stdint.h>

// You may have some issues compiling this to 16 bit. Like getting python to work.
#ifdef \_LP64
#define IS64BIT 1
#else
#define IS32BIT 1
#endif

// Willing to support other compilers, if you can get me access to them, and they
// build python
#ifdef **GNUC**
#define USING\_GCC 1
#elif **clang**
#define USING\_CLANG 1
#else
#error Only support for gcc or clang currently
#error if you port to another compiler, please
#error send back the patch to <https://github.com/stillson/rdrand>
#endif

#if PY\_MAJOR\_VERSION == 2
#define PYTHON2 1
#elif PY\_MAJOR\_VERSION == 3
#define PYTHON3 1
#else
#error requires python 2 or 3
#endif

uint64\_t get\_bits\_using\_rdrand(void);
uint64\_t get\_bits\_using\_rdseed(void);
int RdRand\_cpuid(void);
int RdSeed\_cpuid(void);

PyDoc\_STRVAR(module\_doc, "rdrand: Python interface to Intel and AMD hardware RNG\n");

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
\*

* CHECK CPU
* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\*! \brief Queries cpuid to see if rdrand is supported
\*

* rdrand support in a CPU is determined by examining the 30th bit of the ecx
* register after calling cpuid.
* \return bool of whether or not rdrand is supported
  \*/

/\*! \def RDRAND\_MASK

* The bit mask used to examine the ecx register returned by cpuid. The
* 30th bit is set.
  \*/
  #define RDRAND\_MASK 0x40000000

/\*! \def RDSEED\_MASK

* The bit mask used to examine the ebx register returned by cpuid. The
* 18th bit is set.
  \*/
  #define RDSEED\_MASK 0x00040000

#define \_\_cpuid(x,y,s) asm("cpuid":"=a"(x[0]),"=b"(x[1]),"=c"(x[2]),"=d"(x[3]):"a"(y),"c"(s))

void
cpuid(unsigned int op, unsigned int subfunc, unsigned int reg[4])
{

#if USING\_GCC && IS64BIT
\_\_cpuid(reg, op, subfunc);
#else
asm volatile("pushl %%ebx \n\t" /\* save %ebx */
"cpuid \n\t"
"movl %%ebx, %1 \n\t" /* save what cpuid just put in %ebx */
"popl %%ebx \n\t" /* restore the old %ebx \*/
: "=a"(reg[0]), "=r"(reg[1]), "=c"(reg[2]), "=d"(reg[3])
: "a"(op), "c"(subfunc)
: "cc");
#endif
}

int
RdRand\_cpuid(void)
{
unsigned int info[4] = {-1, -1, -1, -1};

/\* Are we on an Intel or AMD processor? \*/

cpuid(0, 0, info);

if (!(( memcmp((void \*) &info[1], (void \*) "Genu", 4) == 0 &&

memcmp((void \*) &info[3], (void \*) "ineI", 4) == 0 &&

memcmp((void \*) &info[2], (void \*) "ntel", 4) == 0 )

||

( memcmp((void \*) &info[1], (void \*) "Auth", 4) == 0 &&

memcmp((void \*) &info[3], (void \*) "enti", 4) == 0 &&

memcmp((void \*) &info[2], (void \*) "cAMD", 4) == 0 )))

return 0;

/\* Do we have RDRAND? \*/

cpuid(1, 0, info);

int ecx = info[2];

if ((ecx & RDRAND\_MASK) == RDRAND\_MASK)

return 1;

else

return 0;

}

int
RdSeed\_cpuid(void)
{
unsigned int info[4] = {-1, -1, -1, -1};

/\* Are we on an Intel or AMD processor? \*/

cpuid(0, 0, info);

if (!(( memcmp((void \*) &info[1], (void \*) "Genu", 4) == 0 &&

memcmp((void \*) &info[3], (void \*) "ineI", 4) == 0 &&

memcmp((void \*) &info[2], (void \*) "ntel", 4) == 0 )

||

( memcmp((void \*) &info[1], (void \*) "Auth", 4) == 0 &&

memcmp((void \*) &info[3], (void \*) "enti", 4) == 0 &&

memcmp((void \*) &info[2], (void \*) "cAMD", 4) == 0 )))

return 0;

/\* Do we have RDSEED? \*/

cpuid(7, 0, info);

int ebx = info[1];

if ((ebx & RDSEED\_MASK) == RDSEED\_MASK)

return 1;

else

return info[1];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#if IS64BIT

#define GETRAND(rando) asm volatile("1:\n" "rdrand %0\n" "jnc 1b\n" :"=a"(rando) : : "cc")

// read random bytes into supplied buffer
// len is in 64 bit words, not bytes
void
fill\_buf\_using\_rdrand(uint64\_t \*buf, uint32\_t buf\_len)
{
unsigned long int rando = 0;

for (uint32\_t i = 0; i < buf\_len; i++)

{

GETRAND(rando);

buf[i] = rando;

}

}

//utility to return 64 random bits from RdRand
uint64\_t
get\_bits\_using\_rdrand(void)
{
unsigned long int rando = 0;
GETRAND(rando);
return rando;
}
#elif IS32BIT

#define GETRAND(rando) asm volatile("1:\n" "rdrand %0\n" "jnc 1b\n" :"=a"(rando) : : "cc")

// read random bytes into supplied buffer
// len is in 64 bit words, not bytes
void
fill\_buf\_using\_rdrand(uint64\_t \*buf, uint32\_t buf\_len)
{
unsigned int prerand = 0;
uint64\_t prerando = 0;
uint64\_t rando = 0;

for (uint32\_t i = 0; i < buf\_len; i++)

{

GETRAND(prerand);

rando = prerand;

prerando <<= 32;

rando |= prerando;

GETRAND(prerand);

prerando = prerand;

prerando <<= 32;

rando |= prerando;

buf[i] = rando;

}

}

uint64\_t
get\_bits\_using\_rdrand(void)
{
register unsigned int pre\_rand;
union{
uint64\_t rando;
struct {
uint32\_t rando1;
uint32\_t rando2;
} i;
} un;

GETRAND(pre\_rand);

un.i.rando1 = pre\_rand;

GETRAND(pre\_rand);

un.i.rando2 = pre\_rand;

return un.rando;

}

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#if IS64BIT

#define GETSEED(rando) asm volatile("1:\n" "rdseed %0\n" "jnc 1b\n" :"=a"(rando) : : "cc")

// read random bytes into supplied buffer
// len is in 64 bit words, not bytes
void
fill\_buf\_using\_rdseed(uint64\_t \*buf, uint32\_t buf\_len)
{
unsigned long int rando = 0;

for (uint32\_t i = 0; i < buf\_len; i++)

{

GETSEED(rando);

buf[i] = rando;

}

}

//utility to return 64 random bits from RdSeed
uint64\_t
get\_bits\_using\_rdseed(void)
{
unsigned long int rando = 0;
GETSEED(rando);
return rando;
}
#elif IS32BIT

#define GETSEED(rando) asm volatile("1:\n" "rdseed %0\n" "jnc 1b\n" :"=a"(rando) : : "cc")

uint64\_t
get\_bits\_using\_rdseed(void)
{
unsigned int prerand;
union{
uint64\_t rando;
struct {
uint32\_t rando1;
uint32\_t rando2;
} i;
} un;

GETSEED(prerand);

un.i.rando1 = prerand;

GETSEED(prerand);

un.i.rando2 = prerand;

return un.rando;

}

void
fill\_buf\_using\_rdseed(uint64\_t \*buf, uint32\_t buf\_len)
{
for (uint32\_t i = 0; i < buf\_len; i++)
{
buf[i] = get\_bits\_using\_rdseed();
}
}

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* General python code \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static PyObject \*
get\_bits(PyObject \*self, PyObject \*args, uint64\_t quad(void), void fill(uint64\_t \*, uint32\_t))
{
int num\_bits, num\_bytes;
int num\_quads, num\_chars;
unsigned char \* data = NULL;
uint64\_t rando;
unsigned char last\_mask, lm\_shift;
PyObject \*result;

if ( !PyArg\_ParseTuple(args, "i", &num\_bits) )

return NULL;

if (num\_bits <= 0)

{

PyErr\_SetString(PyExc\_ValueError, "number of bits must be greater than zero");

return NULL;

}

num\_bytes = num\_bits / 8;

lm\_shift = num\_bits % 8;

last\_mask = 0xff >> (8 - lm\_shift);

if (lm\_shift)

num\_bytes++;

num\_quads = num\_bytes / 8;

## Copilot

The error messages you're seeing are related to the inline assembly code in your C source file. The push and pop instructions are used in the cpuid function, which is defined in your code. These instructions are used to save and restore the value of the ebx register around the cpuid instruction.

The issue is that the push and pop instructions are expecting a 64-bit operand (register or memory location), but the l suffix is specifying a 32-bit operand. This is causing the assembler to generate an error.

Here's how you can modify the cpuid function to fix this issue:

void

cpuid(unsigned int op, unsigned int subfunc, unsigned int reg[4])

{

#if USING\_GCC && IS64BIT

\_\_cpuid(reg, op, subfunc);

#else

asm volatile("pushq %%rbx \n\t" /\* save %ebx \*/

"cpuid \n\t"

"movl %%ebx, %1 \n\t" /\* save what cpuid just put in %ebx \*/

"popq %%rbx \n\t" /\* restore the old %ebx \*/

: "=a"(reg[0]), "=r"(reg[1]), "=c"(reg[2]), "=d"(reg[3])

: "a"(op), "c"(subfunc)

: "cc");

#endif

}

In this modified version of the function, I've replaced pushl with pushq and popl with popq to specify 64-bit operands. I've also replaced %%ebx with %%rbx, which is the 64-bit version of the ebx register.

Please try making these changes and then compile your code again. If you encounter any more issues, feel free to ask! 😊

Source: Conversation with Bing, 18/5/2024

1. [github.com](https://github.com/intel/kernelflinger/tree/9a84515a0f2b6b5697c49c770513b8c432b6f9d4/libkernelflinger%2Flib.c)