What is a Cell ID?

What is this?

During your pyLCIO event loop, you may have seen something like:

```
col = event.getCollection("ECalBarrelCollection")
enc = col.getParameters().getStringVal(EVENT.LCIO.CellIDEncoding)
dec = UTIL.BitField64(enc)

for hit in col:
    dec.setValue(hit.getCellID0() | (hit.getCellID1() << 32))
    print(dec[“layer”].value())
```

This is how we access encoded/decoded info about a calorimeter cell

i.e. it answers the question: which calorimeter cell am I looking at now?

Let’s look at a few parts of this

Cell ID

```
hit.getCellID0() | (hit.getCellID1() << 32)
```

Each calorimeter cell has 64 bits of identifying information

Helpful quantities like system (ecal, hcal, barrel, endcap) and layer

They can be accessed as two 32-bit integers via `getCellID0()` and `getCellID1()`

Encoding

```
enc = col.getParameters().getStringVal(EVENT.LCIO.CellIDEncoding)
```

In principle, each collection could have different ways of encoding information

e.g. ecal barrel could have different encoding than hcal barrel

→ That’s why encoding is an attribute of the collection

In the example of the previous slide,

```
>>> print(enc) # its a string
‘system:0:5,side:5:-2,module:7:8,stave:15:4,layer:19:9,submodule:28:4,x:32:-16,y:48:-16'
```

However, in practice, calorimeters use same encoding for all events

→ Not necessary to get `enc` on every event loop. Better to define once imho

Encoding

```
>>> print(enc)
'system:0:5,side:5:-2,module:7:8,stave:15:4,layer:19:9,submodule:28:4,x:32:-16,y:48:-16'
```

Field	Offset [bits]	Width [bits]	Integer
system	0	5	Unsigned
side	5	2	Signed
module	7	8	Unsigned
stave	15	4	Unsigned
layer	19	9	Unsigned
submodule	28	4	Unsigned
x	32	16	Signed
y	48	16	Signed

Fields

I’m still understanding the fields myself lol

I’ve used system, side, and layer

Not 100% sure about module, stave, submodule, x, and y

Alex probably needs to understand calorimeter granularity better

lcgeo / MuColl / MuColl_v1.1.3 / **config.xml**

) lines (144 loc) · 7.7 KB

```
    <!-- Detector IDs -->
    <constant name="DetID_VXD_Barrel"     value="1"/>
    <constant name="DetID_VXD_Endcap"     value="2"/>

    <constant name="DetID_IT_Barrel"      value="3"/>
    <constant name="DetID_IT_Endcap"      value="4"/>

    <constant name="DetID_OT_Barrel"      value="5"/>
    <constant name="DetID_OT_Endcap"      value="6"/>

    <constant name="DetID_ECal_Barrel"    value="20"/>
    <constant name="DetID_ECal_Endcap"    value="29"/>

    <constant name="DetID_HCAL_Barrel"    value="10"/>
    <constant name="DetID_HCAL_Endcap"    value="11"/>
    <constant name="DetID_HCAL_Ring"      value="12"/>

    <constant name="DetID_Yoke_Barrel"    value="13"/>
    <constant name="DetID_Yoke_Endcap"    value="14"/>
```

https://github.com/MuonColliderSoft/lcgeo/blob/master/MuColl/MuColl_v1.1.3/config.xml

Decoding

Decoder class (BitField64) can decode 64 bits of cell ID for us

Create a decoder with a particular encoder string; set encoding; access decoded info

```
dec = UTIL.BitField64(enc)

for hit in col:
    dec.setValue( ... )
    print(dec[“layer”].value())
```

Internally, decoder is a map from string to integer (`std::map<std::string, uint>`)

Interally, decoder does bit shifting and masking (no magic)

Decoding

```
>>> print(enc)
'system:0:5,side:5:-2,module:7:8,stave:15:4,layer:19:9,submodule:28:4,x:32:-16,y:48:-16'
```

I can grab a random calorimeter cell from a slcio file and decode it by hand:

```
getCellID0() yields 3671316
getCellID1() yields 4587605
```

```
>>> print(f"{3671316:032b}")
00000000001110000000010100010100
```

MSB

```
00000000010001100000000001010101
```

y	x
70	85

LSB

```
00000000001110000000010100010100
```

submodule	layer	stave	module	side	system
0	7	0	10	0	20

This calorimeter cell is from ecal barrel, layer 7, for example

Decoding

y	x
70	85

subm odule	layer	stave	module	si de	system
0	7	0	10	0	20

Decoding by hand gives same result as BitField64 decoder:

```
>>> print(dec.valueString())
system:20,side:0,module:10,stave:0,layer:7,submodule:0,x:85,y:70
```

Decoded info in action

haster ▾ **MuC-Tutorial** / analysis / other / **make_recoHitPlots_BIB.C**

Blame 293 lines (194 loc) · 8.93 KB

```
    for (int ihit=0; ihit<calo_n; ++ihit){

      // CellID encoding: "system:5,side:-2,module:8,stave:4,layer:9,submodule:4,x:32:-16,y:-16"
      const unsigned int system = (unsigned) ( calo_id0[ihit] & 0x1f );
      const int side = (int) ( (calo_id0[ihit] >> 5) & 0x3 );
```

```
// --- ECAL endcap
if ( system==29  && side==1 ){

  h06_occupancy_ecalE->Fill(calo_y[ihit], calo_x[ihit]);
  h08_depth_ecalE->Fill(calo_z[ihit],calo_y[ihit]);

}
```

Revisiting this

```
col = event.getCollection("ECalBarrelCollection")
enc = col.getParameters().getStringVal(EVENT.LCIO.CellIDEncoding)
dec = UTIL.BitField64(enc)

for hit in col:
    dec.setValue(hit.getCellID0() | (hit.getCellID1() << 32))
    print(dec[“layer”].value())
```

Each collection can have a unique encoding (although they don’t in practice)

The “encoding” is a comma-separate string of names and bit info

The “decoder” is a simple class which holds the encoding string and a map

Each hit has 64 bits of identifying information, accessible as two 32-bit ints

The “decoder” converts those 64 bits into values with simple bit manipulation

Unsolicited Alex commentary

Packing information into integers: good idea for saving space. Lots of calorimeter hits!

Would be cool if getCellID() were 64-bit accessor instead of getCellID0() and getCellID1()

Coding with strings: not very nice. Prone to errors via typos in the strings

Using negative sign to indicate signed int: not very nice. Prefer negative to mean negative

Not necessary to include bit offset in the encoding, imho. Could assume bits are packed

Bonus

const char* LCIO::CellIDEncoding = "CellIDEncoding" ;
https://github.com/MuonColliderSoft/LCIO/blob/master/src/cpp/src/IMPL/LCIO.cc

LCParametersImpl::getStringVal
https://github.com/MuonColliderSoft/LCIO/blob/master/src/cpp/src/IMPL/LCParametersImpl.cc

BitField64
https://github.com/MuonColliderSoft/LCIO/blob/master/src/cpp/include/UTIL/BitField64.h

Two’s complement
https://stackoverflow.com/questions/1604464/twos-complement-in-python

An example decoding by hand

```
def decodeByHand(id0: int, id1: int) -> dict[str, int]:
    def mask(nbits):
        """ e.g. mask(4) returns 0b1111 """
        return (1 << nbits) - 1
    d = {}
    d["system"]    = id0 & mask(5)
    d["side"]      = twos_complement((id0 >> 5) & mask(2), 2)
    d["module"]    = (id0 >> 7) & mask(8)
    d["stave"]     = (id0 >> 15) & mask(4)
    d["layer"]     = (id0 >> 19) & mask(9)
    d["submodule"] = (id0 >> 28) & mask(4)
    d["x"]         = twos_complement(id1 & 0xff, 16)
    d["y"]         = twos_complement(id1 >> 16, 16)
    return d


def twos_complement(val: int, bits: int) -> int:
    """ https://stackoverflow.com/questions/1604464/twos-complement-in-python """
    if (val & (1 << (bits - 1))) != 0: # if sign bit is set e.g., 8bit: 128-255
        val = val - (1 << bits)        # compute negative value
    return val                         # return positive value as is
```

Encoding

```
col = event.getCollection("HCalBarrelCollection")
encoding = col.getParameters().getStringVal(EVENT.LCIO.CellIDEncoding)
decoder = UTIL.BitField64(encoding)
```

```
const char* LCIO::CellIDEncoding = "CellIDEncoding" ;
```

https://github.com/MuonColliderSoft/LCIO/blob/master/src/cpp/src/IMPL/LCIO.cc#L70

LCParametersImpl::getStringVal

https://github.com/MuonColliderSoft/LCIO/blob/master/src/cpp/src/IMPL/LCParametersImpl.cc#L47-L58

https://stackoverflow.com/questions/1604464/twos-complement-in-python

More-human-readable version

```
encoding =
‘system:0:5,side:5:-2,module:7:8,stave:15:4,layer:19:9,submodule:28:4,x:32:-16,y:48:-16'
```

Field	Offset in bits	Width in bits	Integer
system	0	5	Unsigned
side	5	2	Signed
module	7	8	Unsigned
stave	15	4	Unsigned
layer	19	9	Unsigned
submodule	28	4	Unsigned
x	32	16	Signed
y	48	16	Signed

A complete example

```
import pyLCIO
from pyLCIO import EVENT, UTIL
```