Introduction To SixTrackLib Implementation And Design

In this section, we cover the following topics:

- Modelling the Particle State
- Lattice And Beam Elements
- Complementary and Required Externall Libraries & Modules
    - pysixtrack
    - sixtracktools
    - CObjects Buffer
- Logistics of Tracking: Tracking Modes, TrackJob

Modeling the Particle State

In [25]:
```
p = st.Particles(num_particles=1, p0c=6.5e12, q0=1)
```

In [26]:
```
print( f"""Particle-State:
Six main degrees of freedom: x     = {p.x} m\r\n
                             y     = {p.y} m\r\n
                             zeta  = {p.zeta} m\r\n
                             px    = {p.px} rad ( px = Px / P0 )\r\n
                             py    = {p.py} rad ( py = Py / P0 )\r\n
                             delta = {p.delta}     ( δ  = ( P - P0 ) / P0 )\r\n
""")
```

```
Particle-State:
Six main degrees of freedom: x     = [0.] m

                             y     = [0.] m

                             zeta  = [0.] m

                             px    = [0.] rad ( px = Px / P0 )

                             py    = [0.] rad ( py = Py / P0 )

                             delta = [0.]     ( δ  = ( P - P0 ) / P0 )
```

In [29]:

```
print(f"""
4 Logical coordinates: state       = {p.state} ( 1 == active, 0 == lost )\r\n
                       at_element  = {p.at_element}\r\n
                       at_turn     = {p.at_turn}\r\n
                       particle_id = {p.particle_id}\r\n""")
```

```
4 Logical coordinates: state       = [1] ( 1 == active, 0 == lost )

                       at_element  = [0]

                       at_turn     = [0]

                       particle_id = [0]
```

In [30]:

```
print(f"""
5 Attributes of the ref particle: q0     = {p.q0} x proton charge\r\n
                                  mass0  = {p.mass0} eV/c²\r\n
                                  beta0  = {p.beta0} ( β0 = v0/c )\r\n
                                  gamma0 = {p.gamma0} ( γ0 = 1 / √( 1 - β0²) )\
r\n
                                  p0c    = {p.p0c} eV  ( p0c = P0 * c )\r\n
""")
```

```
5 Attributes of the ref particle: q0     = [1.] x proton charge

                                  mass0  = [9.38272081e+08] eV/c²

                                  beta0  = [0.99999999] ( β0 = v0/c )

                                  gamma0 = [6927.62813396] ( γ0 = 1 / √( 1 - β
0²) )

                                  p0c    = [6.5e+12] eV  ( p0c = P0 * c )
```

In [31]:

```
print(f"""
6 auxilliariy coordinates: s            = {p.s} m   ( s ~ distance from begin o
f lattice )\r\n
                           psigma       = {p.psigma}     ( psigma = (E-E0)/(bet
a0*P0*c) )\r\n
                           rpp          = {p.rpp}     ( rpp = P0 / P )\r\n
                           rvv          = {p.rvv}     ( rvv = β / β0 = (v/c) /
β0 )\r\n
                           charge_ratio = {p.charge_ratio}     ( charge_ratio =
q/q0 )\r\n
                           chi          = {p.chi}     ( χ = ( q/q0 ) / ( m/mass
0 ) )\r\n""")
```

```
6 auxilliariy coordinates: s            = [0.] m   ( s ~ distance from begin o
f lattice )
                           psigma       = [0.]     ( psigma = (E-E0)/(beta0*P0
*c) )
                           rpp          = [1.]     ( rpp = P0 / P )
                           rvv          = [1.]     ( rvv = β / β0 = (v/c) / β0
)
                           charge_ratio = [1.]     ( charge_ratio = q/q0 )
                           chi          = [1.]     ( χ = ( q/q0 ) / ( m/mass0
) )
```

There are several ways to get a lattice

1. manually, element by element
2. load from a binary dump
3. import from pysixtrack
4. import from MAD-X (via pysixtrack)
5. import from SixTrack (via pysixtrack)

1. Build Lattice Manually

In [7]:
```
# We can use a st.Elements() class to organize individual beam-elements into a
lattice

lattice = st.Elements()
drift = lattice.Drift(lenth=0.2) # length in [m]
quad  = lattice.Multipole( knl=[0.0, 0.01] ) # ksl ... skew multipole parameter
s

lattice.get_elements()
```

Out[7]:
```
[<Drift at 128
   length:0.0
 >,
 <Multipole at 136
   order:1
   length:0.0
   hxl:0.0
   hyl:0.0
   bal:[0.   0.   0.01 0.  ]
 >]
```

In [8]:
```
# Dump the lattice to a binary file
lattice.to_file( "./demo_lattice.bin" )
del lattice # make sure we start from scratch
```

2. Load From Binary Dump

In [9]:
```
lattice = st.Elements().fromfile( "./demo_lattice.bin" )

lattice.get_elements()
```

Out[9]:
```
[<Drift at 128
   length:0.0
 >,
 <Multipole at 136
   order:1
   length:0.0
   hxl:0.0
   hyl:0.0
   bal:[0.   0.   0.01 0.  ]
 >]
```

3. Import From PySixtrack

Question: What is pysixtrack ?

- pysixtrack is a minimal & straight-forward particle tracking implementation written purely in Python 3
- https://github.com/SixTrack/pysixtrack (https://github.com/SixTrack/pysixtrack)
- Independent of SixTrackLib, easy to understand & extend
- "Playground", Area for prototyping, testing and debugging new physics

In [10]:
```
import pysixtrack as py6tr

# We can use any iterable object to store a lattice:
seq = [ py6tr.elements.Drift(length=0.2), py6tr.elements.Multipole(knl=[0.0, 0.
01]) ]

print( seq )
```

```
[Drift(length=0.2), Multipole(knl=[0.0, 0.01], ksl=[0], hxl=0, hyl=0, length=
0)]
```

In [11]:
```
# Use pysixtrack.Line object to store the sequence of beam-elements
other_lattice = py6tr.Line( seq )

# ensure we start from scratch with lattice
del lattice

# import other_lattice as a SixTrackLib lattice:
lattice = st.Elements().from_line( other_lattice )

print( lattice.get_elements() )
```

```
[<Drift at 128
  length:0.2
>, <Multipole at 136
  order:1
  length:0.0
  hxl:0.0
  hyl:0.0
  bal:[0.   0.   0.01 0.  ]
>]
```

4. Import From MAD-X Using pysixtrack (& cpymad)

The idea is to keep SixTrackLib as minimal as possible.
→ No I/O and Import/Export Helpers in SixTrackLib, use pysixtrack as an intermediate layer

Note: For users of the C/C++ API of SixTrackLib, use binary dumps to import lattices and particle state from MAD-X, SixTrack, ...

Note: Import from MAD-X requires the cpymad cython bindings:
https://github.com/hibtc/cpymad (https://github.com/hibtc/cpymad)

In [12]:

```
# requires cpymad -> https://github.com/hibtc/cpymad
from cpymad.madx import Madx
import sixtracklib as st
import pysixtrack as py6tr

from scipy.constants import e, m_p, c
import numpy as np

# Note: pysixtrack / SixTrackLib and MAD-X use different default units for ener
gies!
p0c = 4.7e9 # p0c = P0 * c ; [p0c] = 1 eV
Etot_in_GeV = np.sqrt( p0c * p0c + ( m_p / e ) ** 2 * c ** 4 ) * 1e-9 # [Etot]
= 1 GeV

mad = Madx(stdout=False)
mad.call( file="./demo_lattice.madx" )
mad.command.beam(particle='proton', energy=str(Etot_in_GeV))
mad.use(sequence="DEMO_LATTICE")

# Use the from_madx_sequence() method of pysixtrack's Line to import DEMO_LATTI
CE
# Also, use the remove_zero_length_drifts() and merge_consequentive_drifts()
# helpers to optimize the lattice for SixTrackLib's preferred way of torage :

imported_lattice = py6tr.Line.from_madx_sequence(
    mad.sequence.DEMO_LATTICE, ).remove_zero_length_drifts(
    inplace=True).merge_consecutive_drifts(inplace=True)

# As before, use imported_lattice to build the SixTrackLib lattice
# that we are actually interested in:
lattice = st.Elements().from_line( imported_lattice )
print( lattice.get_elements() )
```

```
[<Drift at 128
  length:0.2
>, <Multipole at 136
  order:1
  length:0.0
  hxl:0.0
```

5. Import From SixTrack Using pysixtrack (& sixtracktools)

Similar approach, but this time we are importing a more sophisticated Lattice:

Note: sixtracktools is a helper library which runs the SixTrack binary and interprets the output, allowing import data into pysixtrack

https://github.com/SixTrack/sixtracktools (https://github.com/SixTrack/sixtracktools)

In [13]:
```
import sixtracktools
import pysixtrack as py6tr
import sixtracklib as st

# SixTrack input files + helper script to run the SixTrack binary are
# in subdirectory here:

!ls -al ./sixtrack_lhc_no_bb_example/
```

```
insgesamt 4732
drwxrwxr-x 2 martin martin    4096 Jun 16 20:31 .
drwxrwxr-x 6 martin martin    4096 Jun 16 23:18 ..
-rw-rw-r-- 1 martin martin 4373916 Dez  6  2018 fort.16
-rw-rw-r-- 1 martin martin  434684 Dez  6  2018 fort.2
-rw-rw-r-- 1 martin martin   10870 Dez  6  2018 fort.3
-rw-rw-r-- 1 martin martin    6144 Dez  6  2018 fort.8
-rwxrwxr-x 1 martin martin     101 Dez  6  2018 runsix
```

In [14]:

```
six = sixtracktools.SixInput("./sixtrack_lhc_no_bb_example" )
import_lattice = py6tr.Line.from_sixinput( six ).remove_zero_length_drifts(
    inplace=True).merge_consecutive_drifts(inplace=True)

print( f"import_lattice has {len(import_lattice)} elements" )

# Same procedure as usual -> convert import_lattice to a SixTrackLattice
lattice = st.Elements().from_line( import_lattice )

# Create a binary dump for the machine description:
lattice.to_file( "./lhc_no_bb_lattice.bin" )

# Verify the size of the binary dump:
!ls -alh ./lhc_no_bb_lattice.bin
```

```
import_lattice has 18403 elements
-rw-rw-r-- 1 martin martin 3,8M Jun 16 23:19 ./lhc_no_bb_lattice.bin
```

Tracking Examples

Simple Tracking Example (CPU)

In [39]:
```
# Create an initial particle distribution:

beam = st.ParticlesSet()
p = beam.Particles(num_particles=10, p0c=6.5e12)
p.x[:] = np.linspace(-1e-6, +1e-6, p.num_particles)

if p.num_particles <= 16:
    print( f"initial transversal displacement for particles: {p.x}\r\n" )

# Load the lattice from the binary dump we crated earlier
lattice = st.Elements().fromfile("./lhc_no_bb_lattice.bin")
print( f"number of elements in lattice: {lattice.cbuffer.n_objects}")

# What's cbuffer -> Cf. BE Seminar talk for details!
```

```
initial transversal displacement for particles: [-1.00000000e-06 -7.77777778e-
07 -5.55555556e-07 -3.33333333e-07
 -1.11111111e-07  1.11111111e-07  3.33333333e-07  5.55555556e-07
  7.77777778e-07  1.00000000e-06]

number of elements in lattice: 18403
```

In [16]:

```python
# Setup a track-job instance:
job = st.TrackJob( lattice, beam )

# Print particle state before tracking:
if p.num_particles <= 16:
    print( f"at_element before tracking: {p.at_element}" )
    print( f"at_turn    before tracking: {p.at_turn}" )
    print( f"state      before tracking: {p.state}" )
    print( f"x          before tracking: {p.x}" )
```

```
at_element before tracking: [0 0 0 0 0 0 0 0 0 0]
at_turn    before tracking: [0 0 0 0 0 0 0 0 0 0]
state      before tracking: [1 1 1 1 1 1 1 1 1 1]
x          before tracking: [-1.00000000e-06 -7.77777778e-07 -5.55555556e-07 -
3.33333333e-07
 -1.11111111e-07  1.11111111e-07  3.33333333e-07  5.55555556e-07
  7.77777778e-07  1.00000000e-06]
```

In [17]:

```
# Track particles <b>until</b> they are in turn 100
job.track_until( 100 )

# Print particle state after tracking for 100 turns:
if p.num_particles <= 16:
    print( f"at_element after tracking: {p.at_element}" )
    print( f"at_turn    after tracking: {p.at_turn}" )
    print( f"state      after tracking: {p.state}" )
    print( f"x          after tracking: {p.x}" )

# Note: the command is called <tt>track_until</tt> - if we call it again,
# it will have no effect because all particles are alreaedy at turn 100!
# -> we would have to call <tt>track_until( 200 )</tt> to get the desired effec
t
```

```
at_element after tracking: [0 0 0 0 0 0 0 0 0 0]
at_turn    after tracking: [100 100 100 100 100 100 100 100 100 100]
state      after tracking: [1 1 1 1 1 1 1 1 1 1]
x          after tracking: [-9.99845051e-07 -7.77634845e-07 -5.55429506e-07 -
3.33228213e-07
 -1.11030165e-07  1.11165448e-07  3.33359448e-07  5.55552646e-07
  7.77745826e-07  9.99939830e-07]
```

In [18]:

```python
# Let's Loose a particle
p.state[0] = 0
if p.num_particles <= 16:
    print( f"state after manually loosing a particle: {p.state}\r\n" )

# Track until turn 200 and verify the result:
job.track_until( 200 )

if p.num_particles <= 16:
    print( f"at_element after 200 turns : {p.at_element}" )
    print( f"at_turn    after 200 turns : {p.at_turn}" )
    print( f"state      after 200 turns : {p.state}" )
    print( f"x          after 200 turns : {p.x}" )
```

```
state after manually loosing a particle: [0 1 1 1 1 1 1 1 1 1]

at_element after 200 turns : [0 0 0 0 0 0 0 0 0 0]
at_turn    after 200 turns : [100 200 200 200 200 200 200 200 200 200]
state      after 200 turns : [0 1 1 1 1 1 1 1 1 1]
x          after 200 turns : [-9.99845051e-07 -7.77491911e-07 -5.55303462e-07
-3.33123094e-07
 -1.10949224e-07  1.11219787e-07  3.33385573e-07  5.55549726e-07
  7.77713872e-07  9.99879665e-07]
```

Simple Tracking Example (OpenCL, GPU)

First, check whether we have any OpenCL devices and whether SixTrackLib has been compiled with OpenCL support:

In [19]:
```
print( f"SixTrackLib has OpenCL support enabled: {st.config.SIXTRACKLIB_MODULES
['opencl']}")
```

```
SixTrackLib has OpenCL support enabled: True
```

In [20]:
```
!clinfo -l
```

```
Platform #0: Intel(R) FPGA Emulation Platform for OpenCL(TM)
 `-- Device #0: Intel(R) FPGA Emulation Device
Platform #1: Intel(R) OpenCL
 `-- Device #0: Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz
Platform #2: Portable Computing Language
 `-- Device #0: pthread-Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz
Platform #3: Intel(R) OpenCL HD Graphics
 `-- Device #0: Intel(R) Gen8 HD Graphics NEO
Platform #4: Experimental OpenCL 2.1 CPU Only Platform
 `-- Device #0: Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz
```

In [21]:

```
# Re-create the initial state:
beam = st.ParticlesSet()
p = beam.Particles(num_particles=10, p0c=6.5e12)
p.x[:] = np.linspace(-1e-6, +1e-6, p.num_particles)
lattice = st.Elements().fromfile("./lhc_no_bb_lattice.bin")

# Again, create a TrackJob. But this time, we pass the "device" string
opencl_job = st.TrackJob( lattice, beam, device="opencl:1.0" )

print( f"track job instance has architecture {opencl_job.arch_str}" )
```

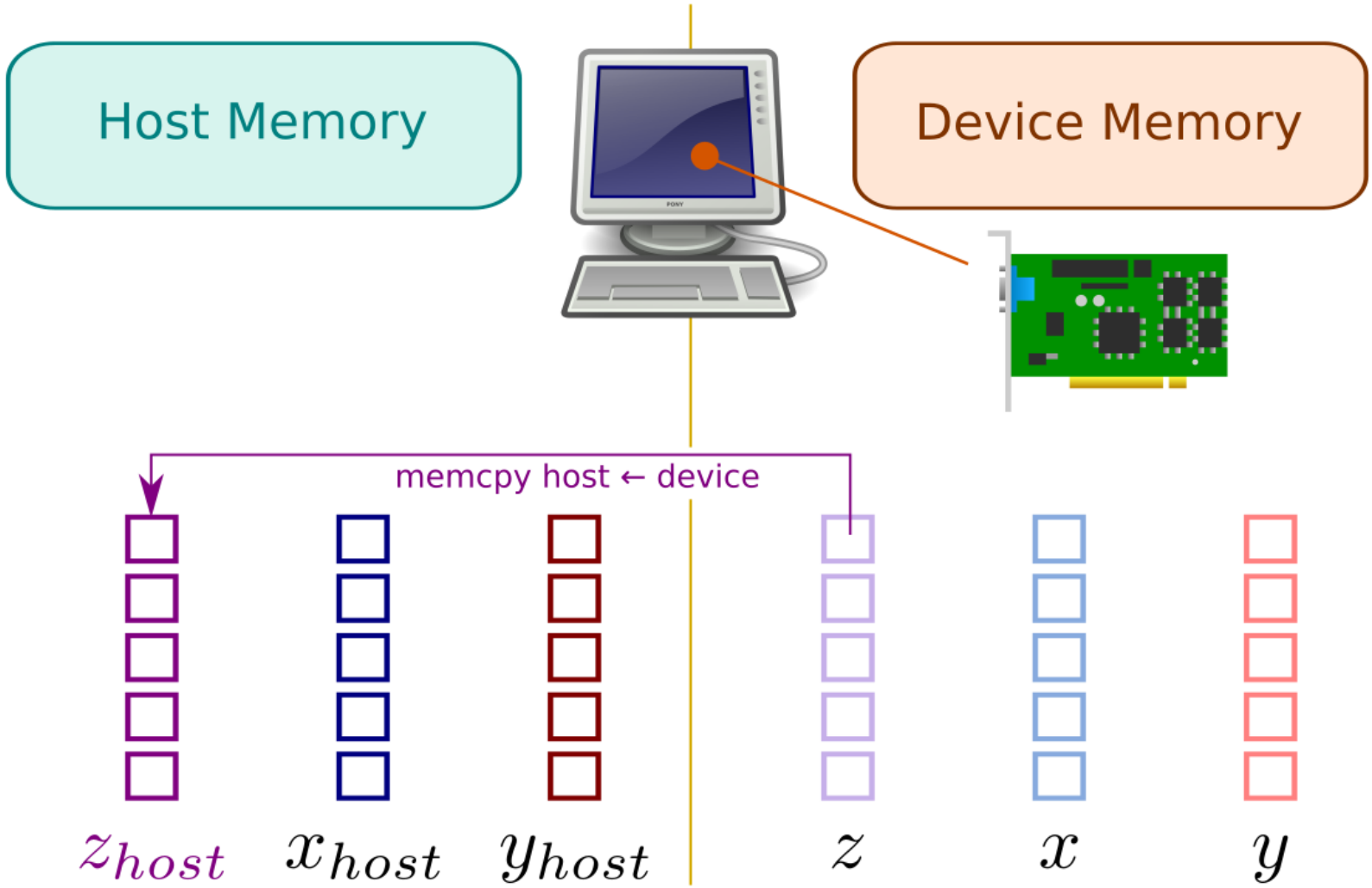
```
track job instance has architecture opencl
```

In [22]:

```
# Again: track until turn 100
opencl_job.track_until( 100 )

# Print particle state after tracking for 100 turns:
if p.num_particles <= 16:
    print( f"at_element after tracking: {p.at_element}" )
    print( f"at_turn    after tracking: {p.at_turn}" )
    print( f"state      after tracking: {p.state}" )
    print( f"x          after tracking: {p.x}" )

# Spoiler: this does not seem to work. Why?
```

```
at_element after tracking: [0 0 0 0 0 0 0 0 0 0]
at_turn    after tracking: [0 0 0 0 0 0 0 0 0 0]
state      after tracking: [1 1 1 1 1 1 1 1 1 1]
x          after tracking: [-1.00000000e-06 -7.77777778e-07 -5.55555556e-07 -
3.33333333e-07
 -1.11111111e-07  1.11111111e-07  3.33333333e-07  5.55555556e-07
  7.77777778e-07  1.00000000e-06]
```

Remember From Introduction To GPU Programming Talk:



Icons: https://openclipart.org - License: Public Domain

In [23]:

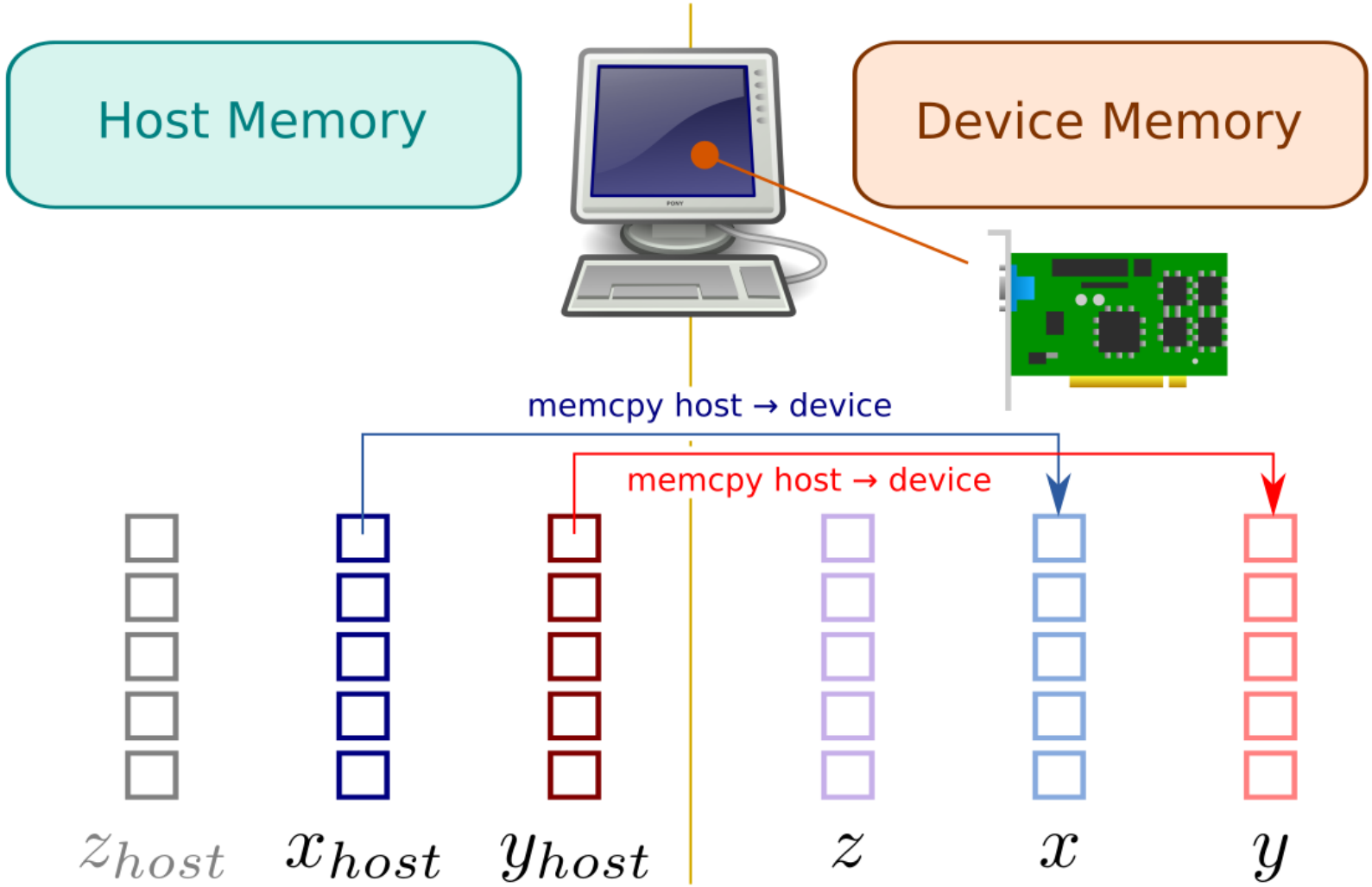
```
# Copying the data back from the device to the host can be a costly operation
# Thus, it is not done automatically -> we have to "collect" the results
opencl_job.collect_particles()

# If we print the the particle data now, we should have the exprected particle
state
if p.num_particles <= 16:
    print( f"at_element after tracking: {p.at_element}" )
    print( f"at_turn    after tracking: {p.at_turn}" )
    print( f"state      after tracking: {p.state}" )
    print( f"x          after tracking: {p.x}" )
```

```
at_element after tracking: [0 0 0 0 0 0 0 0 0 0]
at_turn    after tracking: [100 100 100 100 100 100 100 100 100 100]
state      after tracking: [1 1 1 1 1 1 1 1 1 1]
x          after tracking: [-9.99845052e-07 -7.77634846e-07 -5.55429505e-07 -
3.33228214e-07
 -1.11030165e-07  1.11165447e-07  3.33359448e-07  5.55552645e-07
  7.77745826e-07  9.99939829e-07]
```

When To Call collect_* And push_*:

- Whenever we need the current content of a dataset (e.g. particles, beam elements, ...) on the host, we have to call collect_*
- Whenever we want to send the current content of a dataset (again: particles, beam elements, ...) to the device, we have to call push_*

Host To Device: push_*



Icons: https://openclipart.org - License: Public Domain

In [24]:

```
# Repeat the experiment with the explicitly "lost" particle:
p.state[ 0 ] = 0

if p.num_particles <= 16:
    print( f"state after manually loosing a particle: {p.state}\r\n" )

# It should not be a big surprise, that we need something equivalent to "collec
t"
# but working in the other direction, i.e. from Host -> Device
opencl_job.push_particles()

# Track until turn 200:
opencl_job.track_until( 200 )

# everytime we need the particle state on the host side, we
# have to collect the data:
opencl_job.collect_particles()

# Now, we expect the same output as before:
if p.num_particles <= 16:
    print( f"at_element after 200 turns : {p.at_element}" )
    print( f"at_turn    after 200 turns : {p.at_turn}" )
    print( f"state      after 200 turns : {p.state}" )
    print( f"x          after 200 turns : {p.x}" )
```

```
state after manually loosing a particle: [0 1 1 1 1 1 1 1 1 1]

at_element after 200 turns : [0 0 0 0 0 0 0 0 0 0]
at_turn    after 200 turns : [100 200 200 200 200 200 200 200 200 200]
state      after 200 turns : [0 1 1 1 1 1 1 1 1 1]
x          after 200 turns : [-9.99845052e-07 -7.77491913e-07 -5.55303460e-07
-3.33123095e-07
 -1.10949223e-07  1.11219786e-07  3.33385572e-07  5.55549725e-07
  7.77713871e-07  9.99879664e-07]
```

- collect_* and push_* are potentially expensive calls (band-width for transfer, latency, waiting for all running kernels
- They contribute to t_s !!!
- $\rightarrow$With the exception of the initial push_* when setting up the track job, these are not performed automatically!
- Calling push_* and collect_* has (almost) no negative run-time-cost effect on a CPU track-job
- $\Longrightarrow$ If you call them also with a CPU track-job, your code works on the GPU with just changing the setup line of the track-job!

Generic Tracking Program (Works On CPUs & GPUs With Minimal Changes)

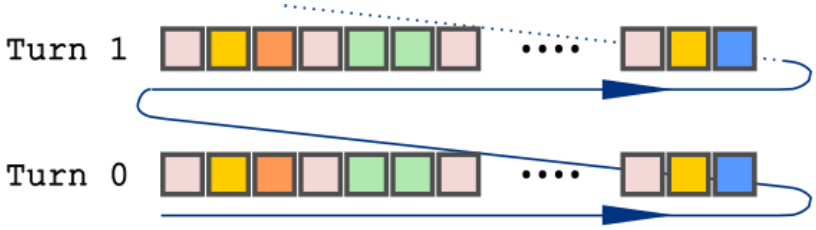
In [38]:

```
beam = st.ParticlesSet()
p = beam.Particles(num_particles=10, p0c=6.5e12)
p.x[:] = np.linspace(-1e-6, +1e-6, p.num_particles)
lattice = st.Elements().fromfile("./lhc_no_bb_lattice.bin")

#device=None  # Or:
device="opencl:0.0" #for GPU

job = st.TrackJob( lattice, beam, device=device )
print( f"Architecture of the track job: {job.arch_str}")

job.track_until( 100 )
job.collect_particles()

p.state[0] = 0 # Mark particle 0 explicitly as lost
job.push_particles()

job.track_until( 200 )
job.collect_particles()

if p.num_particles <= 16:
    print( f"at_element after 200 turns : {p.at_element}" )
    print( f"at_turn    after 200 turns : {p.at_turn}" )
    print( f"state      after 200 turns : {p.state}" )
```

```
Architecture of the track job: opencl
at_element after 200 turns : [0 0 0 0 0 0 0 0 0 0]
at_turn    after 200 turns : [100 200 200 200 200 200 200 200 200 200]
state      after 200 turns : [0 1 1 1 1 1 1 1 1 1]
```

Note: Different Track Nodes

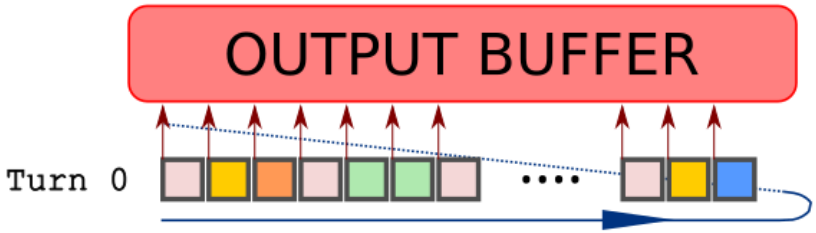
1) track_until Mode:



Track all active particles until they reach at_turn N

```
job.track_until( N )
```

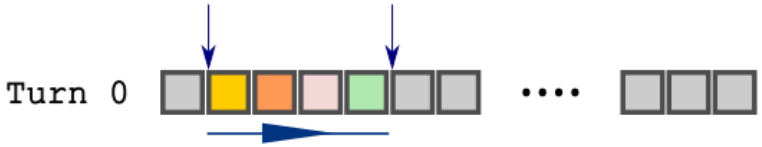
2) track_elem_by_elem Mode:



Like track_until(), but dump (i.e. copy) the particle state to an external buffer before each beam-element

```
job.track_elem_by_elem( N )
```

1) track_line Mode:



Track over subset of lattice

[begin, end)

```
job.track_line( begin, end,
    end_turn=False )
```

End of Interactive Jupyter-Notebook

→ return to main presentation

In [ ]: