UK Atomic Energy Authority

IMAS UDA Workshop

MASTU - IMAS Mappings

+ some JET Mappings

Adam Parker, Jonathan Hollocombe, Stephen Dixon

November 21, 2023 - ITER

Outline

UK Atomic Energy Authority

MASTU - IMAS Mappings

+ some JET Mappings

- MAST-U Mapping Status
- Example output with MAST-U IDSs
- Plugins used
- Example `globals.json`
- Examples of wall, magnetics, and pf_active
- JET Summary IDS
- Writing a plugin

CCFE CULHAM CENTRE FOR FUSION ENERGY

MAST-U Mapping Status

UK Atomic Energy Authority

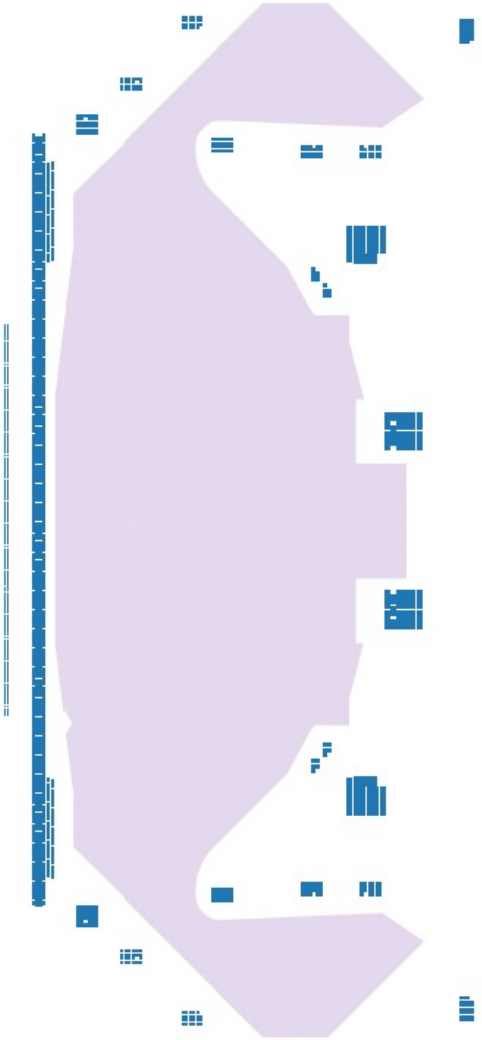
Unmapped	Process started	Developing + Finalising (Progress ongoing)	Mapped and 'producible' Requires validation	Deployed and fully available

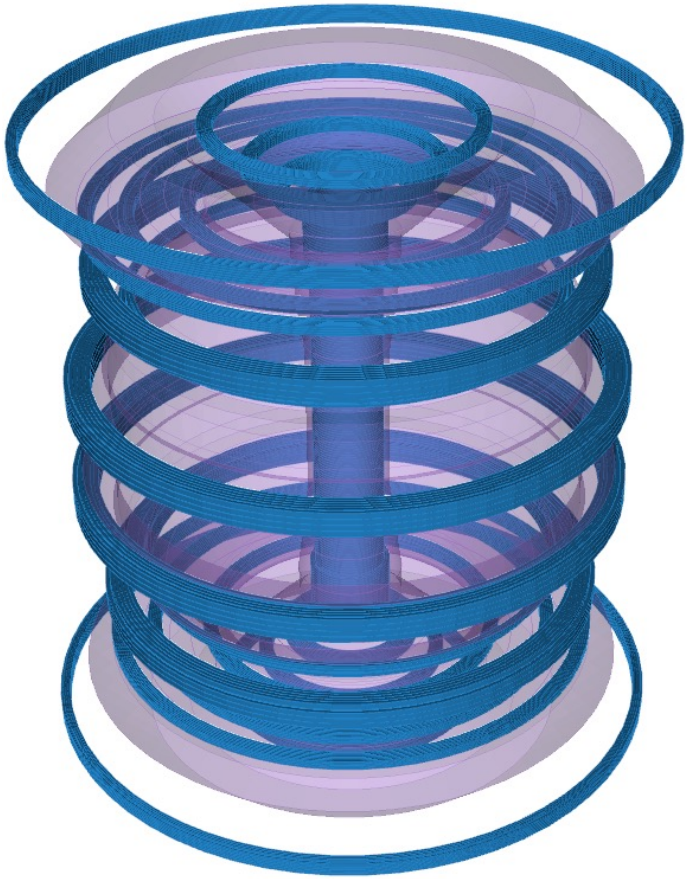
Interface Data Structure (IDS)	Status*	Notes and Comments
magnetics		Reviewed, feedback received
pf_active		Reviewed, feedback received
pf_passive		Reviewed, feedback received
wall		Required signals mapped, handed over for testing
tf		Required signals mapped, handed over for testing
pulse_schedule		Required signals mapped, handed over for testing
mse		Required signals mapped, not yet tested
summary		Partially mapped, not yet tested
nbi		Contacted ROs, initial mappings started
equilibrium/core_profiles		Discussion ongoing (mapping responsibility?)

CCFE CULHAM CENTRE FOR FUSION ENERGY

MAST-U IDS Output

Hot off the press...

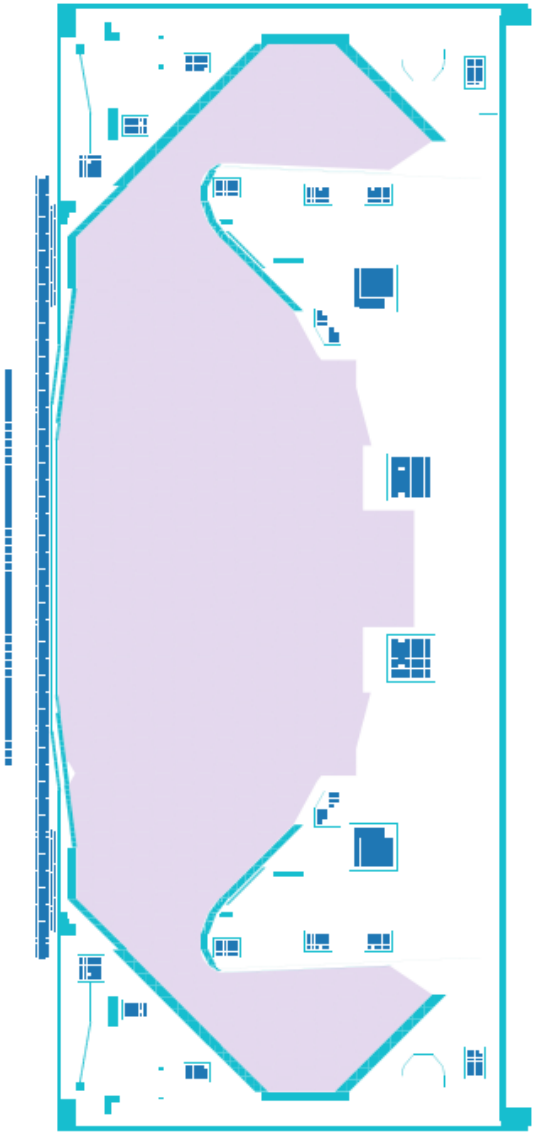


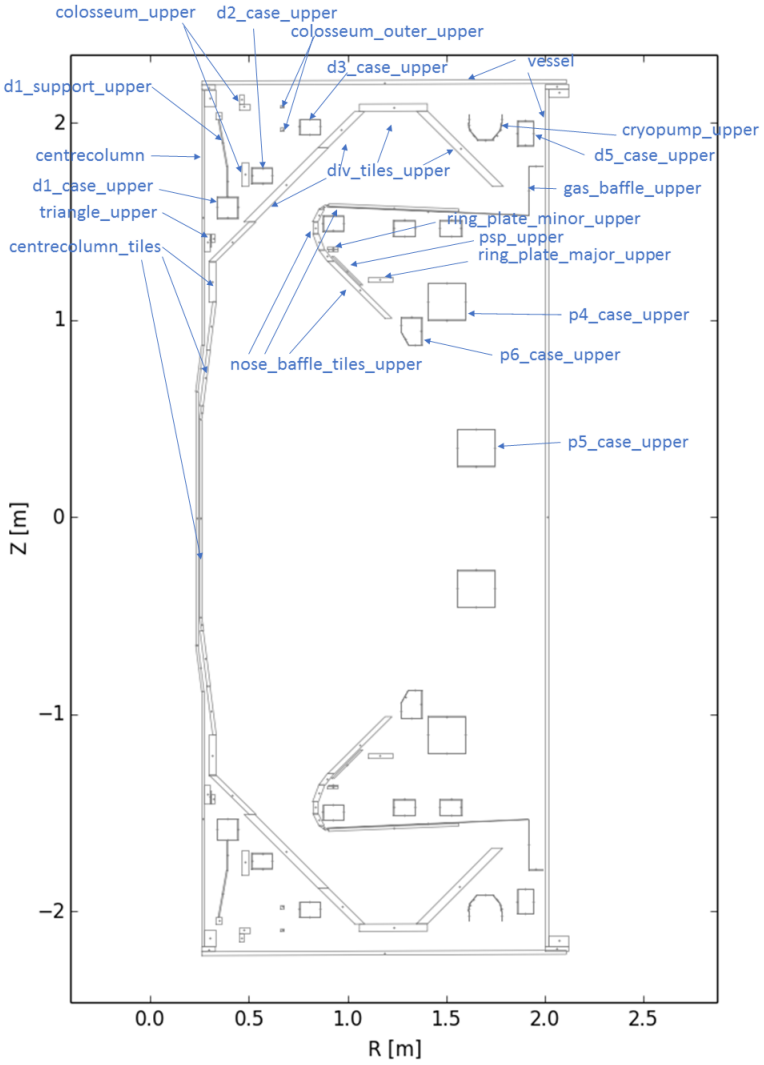


Images provided by Simon Mcintosh

MAST-U IDS Output

Hot off the press...





--- *pf_passive* structures need a little work

MAST-U Plugins for Map Data

UK Atomic Energy Authority

UDA Plugin --> Retrieve MAST-U experimental data

IDS path : **ip[0]/data**

```
UDA::get( signal=/AMC/PLASMA_CURRENT, source=45272, host=uda2.hpc.l, port=56565 )
```

GEOMETRY --> Plugin to map MAST-U machine description data

IDS path : **flux_loop[0]/position[0]/r**

```
GEOMETRY::get(
    signal=/magnetics/d1_upper, key=coordinate.r,
    source=45272, host=uda2.hpc.l, port=56565
)[0]
```

CUSTOM_MASTU --> Plugin for handling custom MAST-U mappings

IDS path : **coil[0]/current/data**

```
CUSTOM_MASTU::pf_coil_current(
    signal=/AMC/ROGEXT/D1U, source=45272,
    host=uda2.hpc.l, port=56565
)
```


CCFE CULHAM CENTRE FOR FUSION ENERGY

PLUGIN_ARGS – Top-level *globals.json*

```
{
    ...
    "PLUGIN_ARGS": {
        "UDA": {
            "source": "{{ shot }}",
            "host": "uda2.hpc.l",
            "port": "56565"
        },
        "GEOMETRY": {
            "source": "{{ shot }}",
            "host": "uda2.hpc.l",
            "port": "56565"
        },
        "CUSTOM_MASTU": {
            "source": "{{ shot }}",
            "host": "uda2.hpc.l",
            "port": "56565"
        }
    }
    ...
}
```

PLUGIN_ARGS defines a dictionary of fields available to a particular plugin by default, do not have to be manually added to each mapping

In this case, all the same. However, only because UDA2 is MAST-U data server for all
Some GEOMETRY or CUSTOM_MASTU mappings actually call UDA::get - pluginception

magnetics IDS – *globals.json*

UK Atomic Energy Authority

Adam: show file

CCFE CULHAM CENTRE FOR FUSION ENERGY

pf_active IDS – *globals.json*

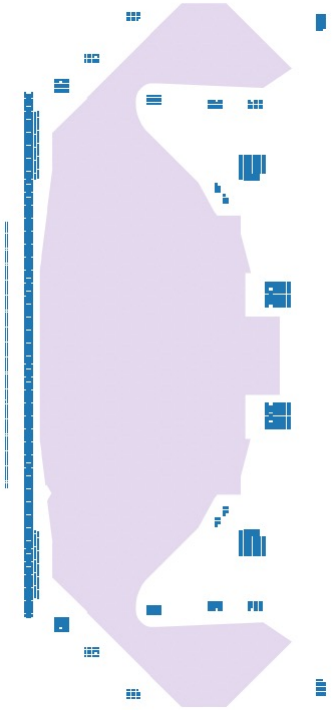
UK Atomic Energy Authority

Adam: show file

CCFE CULHAM CENTRE FOR FUSION ENERGY

wall IDS – *mapping.json*

```
"description_2d[#]/limiter/unit[#]/closed": {
    "MAP_TYPE": "VALUE",
    "VALUE": 1
},
"description_2d[#]/limiter/unit[#]/outline/r": {
    "MAP_TYPE": "PLUGIN",
    "PLUGIN": "GEOMETRY",
    "ARGS": {
        "signal": "/limiter/efit",
        "key": "R"
    }
},
"description_2d[#]/limiter/unit[#]/outline/z": {
    "MAP_TYPE": "PLUGIN",
    "PLUGIN": "GEOMETRY",
    "ARGS": {
        "signal": "/limiter/efit",
        "key": "Z"
    }
}
```



GEOMETRY::get(signal=/limiter/efit, key=R, …)

--> GEOM::get(signal=limiter/efit, Config=1)

WITHIN GEOMETRY PLUGIN
Traverse the returned data structure to R child field

Output array to data block ---------->

```
Attribute outline
        class outline
        Attribute r
        [1.56441581 1.73157942 1.34847534 1.08818924 0.90322602 0.90463918
         0.5341571  0.53829378 0.33279714 0.33279714 0.33479592 0.30311525
         0.30511403 0.26913595 0.27113473 0.26084101 0.26084101 0.27113473
         0.26913595 0.30511403 0.30311525 0.33479592 0.33279714 0.33279714
         0.53829378 0.5341571  0.90463918 0.90322602 1.08818924 1.34847534
         1.73157942 1.56441581 1.37999129 1.37989187 1.19622207 1.19631648
         1.05537415 1.05528355 0.94750249 0.90568638 0.89914298 0.88338786
         0.86768121 0.85132235 0.83348191 0.82606256 0.82267767 0.82102275
         0.82069135 0.82288736 0.827573   0.8391946  0.85524428 0.87756652
         0.89947349 1.18567729 1.27900004 1.296      1.43299997 1.43299997
         1.49000001 1.46000004 1.46000004 1.66100001 1.66100001 1.46000004
         1.46000004 1.49300003 1.47399998 1.43299997 1.43299997 1.296
         1.27900004 1.18567729 0.89947349 0.87756652 0.85524428 0.8391946
         0.827573   0.82288736 0.82069135 0.82102275 0.82267767 0.82606256
         0.83348191 0.85132235 0.86768121 0.88338786 0.89914298 0.90568638
         0.94750249 1.05528355 1.05537415 1.19631648 1.19622207 1.37989187
         1.37999129 1.56441581]
```

magnetics IDS – *mapping.json*

```
{
    ...
    "flux_loop[#]/name": {
        "MAP_TYPE": "VALUE",
        "VALUE": "{{ FL_NAME }}"
    },
    "flux_loop[#]/identifier": {
        "MAP_TYPE": "VALUE",
        "VALUE": "FLUX_LOOP_{{ indices.0 + 1 }}"
    },

    ...
    "flux_loop[#]/flux": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "UDA",
        "ARGS": {
            signal": "/AMB/{{ FL_TYPE }}/{{ FL_NAME }}"
        }
    },
    "flux_loop[#]/voltage": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "UDA",
        "ARGS": {
            signal": "/AMB/LOOPV/{{ LOOPV_NAME }}"
        }
    },
    ...
}
```

Index # = 0

flux_loop[0]/name
"F_BL_01"

flux_loop[0]/name
"FLUX_LOOP_1"

flux_loop[0]/flux
Signal --> /AMB/FLUX/F_BL_01
UDA::get(signal=/AMB/FLUX/F_BL_01, …)
<private data>

flux_loop[0]/voltage
Signal --> /AMB/LOOPV/LV_BL_01
UDA::get(signal=/AMB/LOOPV/LV_BL_01, …)
<private data>

magnetics IDS – *mapping.json*

```
{
    ...
    "ip[#]": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "UDA",
        "ARGS": {
            signal": "/AMC/PLASMA_CURRENT"
        },
        "SCALE": "{{ UNIT_SF }}"
    },
    ...
}
```

magnetics IDS – *mapping.json*

```
{
    ...
    "b_field_pol_probe[#]/position/r"": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "GEOMETRY",
        "ARGS": {
            signal": "{{ BPOL_NAME }}",
            key": "coordinate.r"
        }
    },
    "b_field_pol_probe[#]/position/z"": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "GEOMETRY",
        "ARGS": {
            signal": "{{ BPOL_NAME }}",
                      key": "coordinate.z"
        }
    },
    ...
}
```

```
[ data ]
|-> signal_type
|-> name_
|-> refFrame
|-> type
|-> array
|-> status
|-> version
|-> bandwidth
|[ orientation ]
||-> measurement_direction
|||[ unit_vector ]
|||-> r
|||-> z
|||-> phi
|[ coordinate ]
||-> r
||-> z
||-> phi
|[ geometry ]
||-> length
||-> nturnsLayer1
||-> nturnsLayer2
||-> nturnsLayer3
||-> nturnsLayer4
||-> nturnsTotal
||-> areaLayer1
||-> areaLayer2
||-> areaLayer3
||-> areaLayer4
||-> areaAve
```

`GEOMETRY::get(signal='b_bl1_n02', key=coordinate.z …)`
Data structure received from `'signal = b_bl1_n02'`

Traverse the object to access `'coordinate.z'`

magnetics IDS – *mapping.json*

```
{
    ...
    "b_field_pol_probe[#]/poloidal_angle": {
        "MAP_TYPE": "EXPR",
        "PARAMETERS": {
            "Z": "_pickup[#]/unit_vector/Z",
            "R": "_pickup[#]/unit_vector/R"
        },
        "EXPR": "2*PI-atan2(Z,R)"
    },
    ...
     "_pickup[#]/unit_vector/Z": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "GEOMETRY",
        "ARGS": {
            "signal": "{{ BPOL_NAME }}",
            "key": "orientation.unit_vector.z"
        }
     },
     "_pickup[#]/unit_vector/R": {
        "MAP_TYPE": "PLUGIN",
       "PLUGIN": "GEOMETRY",
        "ARGS": {
            "signal": "{{ BPOL_NAME }}",
            "key": "orientation.unit_vector.r"
        }
     },
    ...
}
```

1) Access the two 'hidden' signals and load into memory

GEOMETRY::get(signal='b_bl1_n02', key=coordinate.z …)
Again traverse to `coordinate.z`

2) Use as input to the expression and evaluate

$$2\pi \times \mathrm{atan2}(Z, R)$$

3) Return result to the DataBlock

```
>>> ids.b_field_pol_probe[0].poloidal_angle
    4.668756008148193
```

pf_active IDS – *mapping.json*

UK Atomic Energy Authority

```
"coil[#]/element[#]/geometry/rectangle/width": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "GEOMETRY",
        "ARGS": {
            "signal": "/magnetics/pfcoil/{{ PF_GEOM_NAME }}",
            "key": "geom_elements.dR"
        },
        "SLICE": "[{{ indices.1 }}]"
    },
    "coil[#]/element[#]/geometry/rectangle/height": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "GEOMETRY",
        "ARGS": {
            "signal": "/magnetics/pfcoil/{{ PF_GEOM_NAME }}",
            "key": "geom_elements.dZ"
        },
        "SLICE": "[{{ indices.1 }}]"
    },
```


CCFE CULHAM CENTRE FOR FUSION ENERGY

pf_active IDS – *mapping.json*

UK Atomic Energy Authority

```
"coil[#]/current" : {
    "MAP_TYPE": "PLUGIN",
    "PLUGIN": "CUSTOM_MASTU",
    "ARGS": {
        "signal": "/AMC/ROGEXT/{{ PF_NAME }}"
    },
    "FUNCTION": "pf_coil_current",
    "SCALE": "{{ UNIT_SF }}"
},
```

Coil[#]/current

Should be trivial no? However, instead of just taking the ROGEXT signal straight, P1 coil needs to have the current *0.5, not so trivial

Using special CUSTOM_MASTU plugin with ::pf_coil_current() functionc

CCFE CULHAM CENTRE FOR FUSION ENERGY

pf_active IDS – *mapping.json*

UK Atomic Energy Authority

```
"circuit[#]/connections" : {
    "MAP_TYPE": "PLUGIN",
    "PLUGIN": "CUSTOM_MASTU",
    "ARGS": {
        "ps_name": "{{ PF_PS_NAME }}"
    },
    "FUNCTION": "pf_conn_matrix"
},
```

Circuit[#]/connections

Another special CUSTOM_MASTU plugin with ::pf_conn_matrix() function essentially to read the circuits connection matrix for MAST-U from JSON

CCFE CULHAM CENTRE FOR FUSION ENERGY

JET summary IDS – *mapping.json*

UK Atomic Energy Authority

JET pulses summarised into **C**entral **P**hysics **F**ile (CPF)
Database of high-level quantities – sampled to identified ROIs

(substantial interpolation methods used)

```
{
    ...
    "global_quantities/ip": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "JETCPF_Reader",
        "ARGS": {
            "signal": "MAGN/IPLA"
        }
    },
    "global_quantities/r0": {
        "MAP_TYPE": "VALUE",
        "VALUE": 2.96
    },
    "global_quantities/b0": {
        "MAP_TYPE": "PLUGIN",
        "PLUGIN": "JETCPF_Reader",
        "ARGS": {
            "signal": "MAGN/BVAC"
        }
    },
    ...
}
```

Simple mapping taken straight from CPF without modification for several summary quantities

Variables can be added to CPF on request
(Derived for PPF)

```
JETCPFReader::get(signal=MAGN/IPLA, pulse=99514, …)
```

Slice to get 2nd element
Array signal: [1.3, 4.5, 3.3] - returns 4.5

CCFE CULHAM CENTRE FOR FUSION ENERGY

JET CPF Reader – *plugin*

UK Atomic Energy Authority

CPF data available by common API

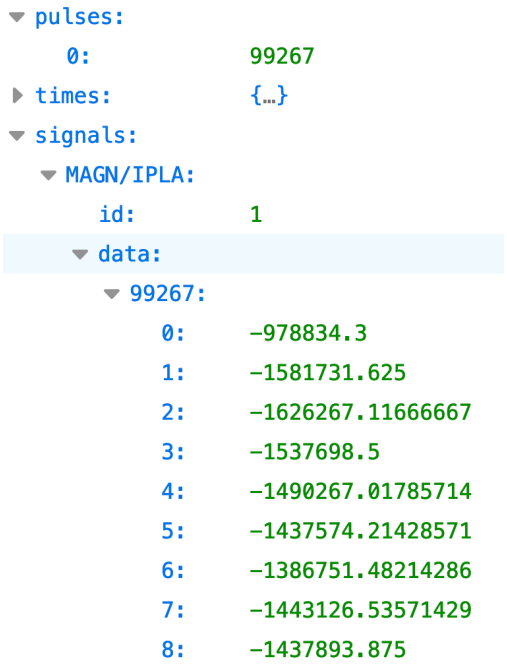
JET_CPF_Reader Plugin created to construct URIs and send requests for CPF data

Example Request (UKAEA firewall / network for access)

`http://data-devel.jet.uk/cpf/extension/api/data?signal=MAGN,IPLA&pulse=99267`

Plugin workflow

1) Interpret http request,
2) Handle response and parse
3) Inspect rank and type
4) Deposit data on DataBlock



CCFE CULHAM CENTRE FOR FUSION ENERGY

Writing a Plugin to Retrieve Data

UK Atomic Energy Authority

1) **Parse request parameters**
   Handle everything that was passed, use as you see fit

2) **Access data**
   Actually read the data, however your experiment does it

3) **Determine rank and type**
   Need to know the shape of the data

4) **Appropriately put on DataBlock**
   Using the known shape of the data, call the correct macro

CCFE
CULHAM CENTRE FOR FUSION ENERGY

Plugin: Return Data Macros

UK Atomic Energy Authority

Just to note, UDA has helper macros to make it easier to return data to the DataBlock

```
SetReturnDataIntScalar(data_block, 23, nullptr)

SetReturnDataFloatScalar(data_block, 4.5, nullptr)

std::vector<int> my_vec = {1, 2, 3};
const size vec_size = my_vec.size();
SetReturnDataIntArray(data_block, 4.5, nullptr);

std::vector<int> my_vec = {2.3, 4.5, 7.6, 5.3};
const size vec_size = my_vec.size();
SetReturnDataIntArray(data_block, my_vec.data(), 1, &vec_size, nullptr);
```


CCFE CULHAM CENTRE FOR FUSION ENERGY

DRAFT Data (JSON)

Easier to see if we just inspect live

UK Atomic Energy Authority

Thanks for listening
Questions / Comments?

Start Day 2 Hands-on 2

CCFE
CULHAM CENTRE FOR FUSION ENERGY