OpenCPI - Open Component Portability Infrastructure

An Open Source Infrastructure for Heterogeneous/Embedded Component-Based Systems and Applications

OpenCPI RCC Development

RCC Development Overview

- RCC Development Overview
- Advanced RCC Features and Functionality

RCC Development Overview

- RCC – Resource-constrained C/C++
- C and C++ language workers
- Focus on C++ moving forward
- "Supported Systems and Platforms" in the *OpenCPI Installation Guide* and *OpenCPI User Guide* lists development hosts, embedded systems platforms

Application Worker Build Steps

1. OPS: Use a pre-existing OPS or create a new one.
2. OCS: Use a pre-existing OCS or create a new one.
3. Create a new worker (modify OWD and source code).
4. Build the worker for the target device(s).
5. Create the unit test (<component>-test.xml, generate, verify and view scripts).
6. Build the unit test.
7. Run the unit test.

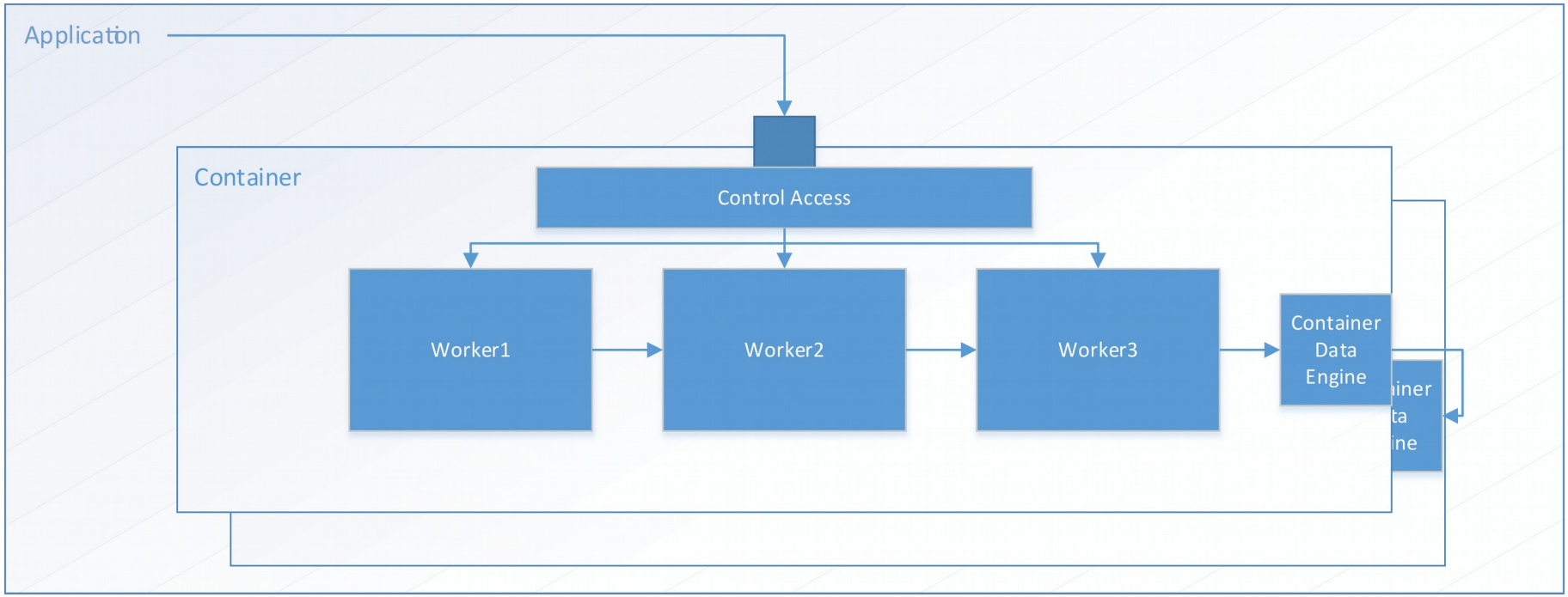
OpenCPI Tutorial 3 demonstrates this process.

Execution Model

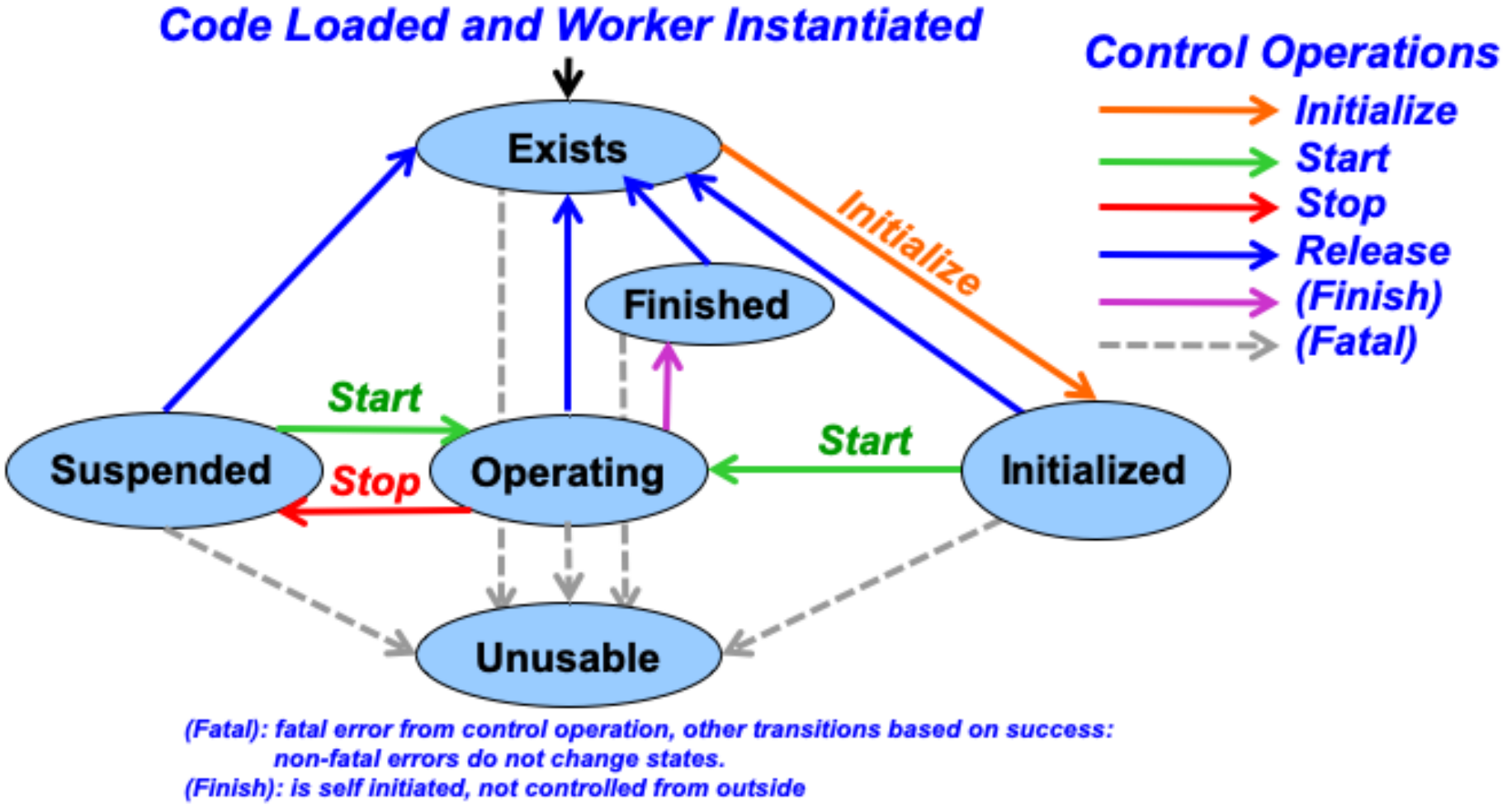
- *Software containers* provide the execution environment for workers
  - Load, execute, control, and configure workers
  - Move data between workers
- Provide interfaces for local services
  - Including software watchdog timer
- Evaluate data availability
  - Consumer's buffer has data
  - Producer's buffer has space
- Each RCC container is a single *thread*

Execution Model (cont.)

- Shared buffers between workers that are in the same container



Worker Life Cycle



(Fatal): fatal error from control operation, other transitions based on success: non-fatal errors do not change states.
(Finish): is self initiated, not controlled from outside

Run Method

- Called by container when all ports are ready (input ports have data and output ports have a free buffer)
- Other advanced use cases; for example, timeouts if run condition changed from default
- Return values to container
  - `RCC_ADVANCE` (good state + advance all ports)
  - `RCC_OK` (good state)
  - `RCC_FINISHED` (finished state)
  - `RCC_ADVANCE_FINISHED` (finished + advance all ports)
  - `RCC_ERROR` (bad state: recoverable; still operating)
  - `RCC_FATAL` (bad state: **unrecoverable**)

Start/Initialize/Stop/Release

OWD XML:

```
<RCCWorker language="c++"
spec="myname_spec.xml"
ControlOperations="start,
stop, initialize,
release"/>
```

C++:

```
RCCResult start() // others
are
                    //
initialize(),
                    // stop(),
and
                    //
release()
{
    //your code goes here
    return RCC_OK;
}
```

Property Access

OCS XML:

```
<property name="myprop"
type="bool"
writeable="true"
volatile="true"/>
```

C++:

```
if (properties().myprop ==
true)
{
  //do something
}

properties().myprop = false;
```

Port Access: Scalar Length

OCS XML:

```
<Port Name="myin"
Producer="false"
Protocol="my_protocol"/>
<Port Name="myout"
Producer="true"
Protocol="my_protocol"/>
```

OPS XML:

```
<Protocol>
<Operation name="myOp">
  <Argument name="myArg1"
    type="bool"/>
 <Operation/>
<Protocol/>
```

Input ports can be assumed based on the protocol, because there is no variable length data. Output port lengths do not need to be set as long as all arguments are scalar:

C++:

```
//Nothing to do
```

Port Access: Scalar Data

OCS XML:

```
<Port Name="myin"
Producer="false"
Protocol="my_protocol"/>
<Port Name="myout"
Producer="true"
Protocol="my_protocol"/>
```

OPS XML:

```
<Protocol>
 <Operation name="myOp">
  <Argument name="myArg1"
type="bool"/>
  <Argument name="myArg2"
type="bool"/>
 <Operation/>
<Protocol/>
```

C++:

```
const bool inArg1  =
myin.myOp().myArg1();
const bool inArg2  =
myin.myOp().myArg2();
myout.myOp().myArg1() = false;
myout.myOp().myArg2() = true;
```

Port Access: Sequence Length

OCS XML:

```
<Port Name="myin"
Producer="false"
Protocol="my_protocol"/>
<Port Name="myout"
Producer="true"
Protocol="my_protocol"/>
```

OPS XML:

```
<Protocol>
<Operation name="myOp">
  <Argument name="myArg1"
   type="bool"
   SequenceLength="2048"/>
 <Operation/>
<Protocol/>
```

C++:

```
//length in elements
const size_t length =
  myin.myOp().myArg1().size();

myout.myOp().myArg1().resize(length);

//length in bytes
myout.setDefaultLength(2*sizeof(bool));
```

Port Access: Sequence Data

OCS XML:

```
<Port Name="myin"
Producer="false"
Protocol="my_protocol"/>
<Port Name="myout"
Producer="true"
Protocol="my_protocol"/>
```

OPS XML:

```
<Protocol>
<Operation name="myOp">
  <Argument name="myArg1"
type="bool"
   SequenceLength="2048"/>
 <Operation/>
<Protocol/>
```

C++:

```
const bool* inData  =
  myin.myOp().myArg1().data();

bool*       outData =
  myout.myOp().myArg1().data();

for (int i = 0; length > i; i++)
{
  //do something with the data
  inData++;
  outData++;
}
```

Port Access: OpCode

OCS XML:

```
<Port Name="myin"
Producer="false"
Protocol="my_protocol"/>
<Port Name="myout"
Producer="true"
Protocol="my_protocol"/>
```

OPS XML:

```
<Protocol>
 <Operation name="MyOp1">
  <Argument name="myArg1"
type="bool"/>
  <Argument name="myArg2"
type="bool"/>
 <Operation/>
 <Operation name="MyOp2">
  <Argument name="otherArg1"
type="bool"
   SequenceLength="2048"/>
 <Operation/>
<Protocol/>
```

C++:

```
//for all outputs on this port
myout.setDefaultOpCode(My_protocolMyOp1_OPE
RATION);
//for the current output on this port
myout.setOpCode(My_protocolMyOp1_OPERATION
);

switch(myin.opCode())
{
  case My_protocolMyOp1_OPERATION:
  {
     //do something
     break;
  }
  case My_protocolMyOp2_OPERATION:
  {
    //do something different
    break;
  }
  default:
  {
    cout << "bad Opcode" << endl;
    break;
  }
}
```

Port Access: Manually Advancing

OCS XML:

```
<Port Name="myin" Producer="false"
Protocol="my_protocol"/>
<Port Name="myout" Producer="true"
Protocol="my_protocol"/>
```

C++ advance all:

```
return RCC_ADVANCE;
```

C++ advance manually:

```
myin.advance();
myout.advance();

return RCC_OK;
```

Local Worker Variables

Define any local variables that you need as class variables.

OWD XML:

```
<RccWorker spec="myworker-
spec.xml" language="C++">
</RccWorker>
```

C++:

```
class My_workerWorker :
  public My_workerWorker base
{

  bool myLocalVar1;
  bool myLocalVar2;

  …

  RCCResult run()
  {
     //use local variables
  }
```

Version 2 for RCC Workers

OWD XML:

```
<RCCWorker language="c++"
spec="myname_spec.xml"
version="2"/>
```

We recommend that all newly created workers use the latest interface version.

version 1:
interprets a zero length message(ZLM) with opcode 0 as end of file(eof) on both input and output ports

version 2:
eof is received on an input port:

```
myin.eof()
```

eof is set on an ouput port:

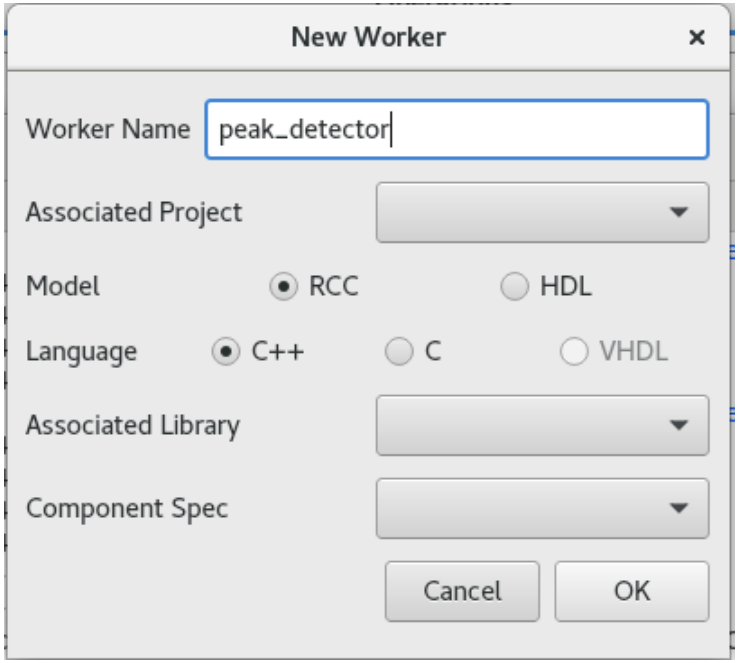
```
myout.setEOF()
```

Creating RCC Workers

With ocpidev:

```
ocpidev create worker peak_detector.rcc
```

With OpenCPI GUI:



Files created:

```
peak_detector.rcc/
|-- gen
|    |-- peak_detector_map.h
|    |-- peak_detector-build.xml
|    |-- peak_detector-skel.cc
|    |-- peak_detector-skel.cc.deps
|    |-- peak_detector-worker.hh
|    |-- peak_detector-worker.hh.deps
|-- peak_detector.cc
|-- peak_detector.xml
```

Cross Building

- An example of a cross-building target is Zynq-arm, the CPU inside the Zynq system-on-chip (SoC).
- To enable cross build:

```
ocpidev build --rcc --rcc-platform xilinx19_2_aarch32
```

- To enable build for x86:

```
ocpidev build --rcc
```

Or use the OpenCPI GUI

RCC Build Artifacts

- A built RCC worker is a Linux shared object file
  - Dynamically loadable on the platform that it is built for
- Info about the worker (XML) embedded in .so

```
ocpixml get myworker.so
```

Advanced RCC Features and Functionality

- RCC Development Overview
- Advanced RCC Features and Functionality

Parameter Access

OWD XML:

```
<RccWorker spec="myworker-
spec.xml"
  language="C++">
  <property name="myparam"
type="bool"
   parameter="true">
</RccWorker>
```

C++:

```
if (MYWORKER_MYPARAM == true)
{
  //do something
}
```

Property Write Notification

OWD XML:

```
<SpecProperty
name="myprop" type="bool"
writeSync="true"/>
```

C++:

```
RCCResult myprop_written()
{
  //your code
  return RCC_OK;
}
```

Property Read Notification

OWD XML:

```
<SpecProperty
name="myprop" type="bool"
readSync="true"/>
```

C++:

```
RCCResult myprop_read()
{
  //your code
  return RCC_OK;
}
```

Optional Ports

OCS XML:

```
<Port Name="myout"
Producer="true"
  optional="true"

Protocol="rstream_protocol"/>
```

C++:

```
if (myout.isConnected())
{
  //output the data to the
port
}
```

RCC Worker as a Control Proxy

- A C++ worker that can manipulate other workers' properties
  - Slave worker can be HDL or another RCC
  - Can be 1:1 or 1:many (workers and/or properties)

- Main use case is for platform development to simplify the user interface when developing a Board Support Package (BSP)

Running a Worker in a Debugger

- Described in the *RCC Development Guide*:
  - Gives directions on how to break `gdb` in your worker when trying to debug
  - Can be used in a standalone graphical debugger such as DDD
  - `gdb` is available native on remote systems (e310); remote debugging with `gdbserver` is not required