POETS

Orchestrator Internals

POETS Orchestrator architecture

Design notes

Volume III - Orchestrator internals

Ver 01	*27 November 2018*
Ver 02	*3 July 2019*
Ver 03	*6 August 2020*
Ver 04	*22 October 2020*

Andrew Brown, Graeme Bragg, Mark Vousden

Documentation structure

Volume I : Introduction
The big picture.

Volume II : Application definition
The interface between the domain-specific front ends and the Orchestrator XML file specification.

Volume III : The Orchestrator (internals)
Internal structure and function of the domain-agnostic software layer.

Volume IV : The Orchestrator (user-facing)
User documentation, and its interaction with the Orchestrator internals.

Volume V : Associated tools and capabilities
Extra functionality overlaid upon the Orchestrator (simulation) plus stand alone graph and test circuit generators.

Volume A : Appendices
Lists, tables enumerated constants, structure layouts and details, worker class documentation.

Revision history

Revision	Date	Description
Ver 02	3 July 2019	Added Section 12: Motherships; section 13: Deploying; Section 14: Application execution. New section 2.3.1.
Ver 03	6 August 2020	OrchBase class description re-written
Ver 04	22 October 2020	Fluid sections replaced with pointers to other documents; these are in **arial black**

This is an evolving set of documents, and the level of detail varies wildly throughout, from syntax diagrams to "...and then a miracle occurred....".

Throughout the lifetime of the project, the intention is that more and more details will be included, things will change, and the miracles will morph into hard quantitative details.

New stuff will appear, almost as if it had been intended to be there all along.

This is an Orchestrator design document. It is not an overall POETS design, or a hardware design, or a front-end design document. These require higher-order magic.

POETS Orchestrator architecture

Design notes

Volume III - Orchestrator internals

1 Introduction

The Orchestrator is a highly asymmetric, multi-threaded (pThreads) heterogeneous MPI program written in C++98. Aside from the limitations on data space, it can be compiled in a 32- or 64-bit environment.

- Most of the major component names in POETS start with a 'P'; this is not to be confused with the multi-threaded components that use pThreads. There's even a POETS class called P_thread, (which has nothing to do with pThreads) but we simply could not think of a sensible alternative name. It's not that confusing really.

- The SoftSwitch is cross-compiled to 32 bit RISC-V binaries.

- C++98 because there is much value in having code pass through as many compilers as possible, and the support for much of C++11/14 is not as widespread as one might wish. Also, my favourite development environment is Borland CBuilder 6.0, which predates C++11. I (usually) try to pass everything through CBuilder and the latest Visual Studio under Windows and gcc under Linux (although the various versions of gcc do not appear to be consistent: gcc 4.2 and 7.0 have differing ideas about what constitutes legal C++.)

- pThreads because it is lightweight, simple and mature; C++11 threads are none of these.

- MPI because, realistically, it is the only game in town unless we want to descend to bare-metal socket programming. MPI under Windows is no longer supported by the MPI consortium; it has been handed off to Microsoft, and the quality of the implementation is what one might expect from Microsoft.

Each process is entered via a `main()` that does little but instantiate a single appropriate class and catch *out of memory, unexpected* and some *unrecoverable error* exceptions. Microsoft MPIexec does not hand console-launched exceptions (signals: ctl-C) into any of the MPI processes; they are trapped by MPIexec and not forwarded. MPIexec catches ctl-C and (usually) aborts (most of) the universe, sometimes pausing to create immortal processes, which are hard to remove.

The principal class of a process and the corresponding process itself are referred to by the same name: for example, the LogServer process is started from a program in a file both called LogServerMain(.cpp), which instantiates a dynamic object called LogServer; most of the action takes place in the principal class constructor.

Source layout in files:
Most classes are defined in their own `*.cpp|.h` files; broadly, the layout is {minor thread code | constructors | destructors | methods *in alphabetical order* | subclasses}.
By order of Micro$oft, templated classes have templated methods and non-templated methods in separate compilation units. I don't know why.

Self-documenting code is a myth propagated by people who are too lazy to write documentation and/or who write code that is so bad they are justly embarrassed by it and do not want anyone to use it. This volume outlines the essential datastructures and major pathways through the Orchestrator; it will be convenient to have the source to hand to refer to details as necessary; even better, have a debugger attached to the executable and step through bits that you need to understand further. Working stuff out from the source alone is not going to work. I wrote it and I can't do that.

The goal of this document is to allow folk "skilled in the art" to get inside the Orchestrator, fix it when bugs are discovered and enhance its capabilities, *whilst understanding what they are doing*. It follows that enhancements to the functionality of the system will be accompanied by extensions and additions to this document (and others where appropriate).

Each non-trivial class is provided with a **`Dump()`** method, that pretty-prints the local data structure (state). The Orchestrator runs under MPI; as a consequence pretty much every print statement needs to be succeeded by a buffer flush if the output is to appear on the screen usefully.

Most of the major classes composing the Orchestrator are written *for* the Orchestrator, and are documented here. They will also contain errors, because they are new. However, there are a fair number of quite sophisticated classes - that were developed in other projects - that are heavily used and that have no knowledge of POETS and are agnostic to its structure. These are called "**generics**". They are all reasonably "battle hardened" - the oldest has been in use for over twenty years - but that doesn't mean they are bug-free or can't be improved. These are documented separately.

2 Interprocess communications

The Orchestrator is a heterogeneous, hybrid MPI system, that conforms to the MPI 2 specification and uses only a small subset of that. It is also asynchronous, and whilst there is nothing even close to the edge of the MPI *specification* there, it is a highly unusual configuration for MPI. Whilst it works reliably - and has been stressed to destruction in numerous ways - it has been an interesting development trajectory, and it is not unlikely that some issues remain undiscovered. Ceaseless vigilance is not enough. Only the paranoid survive.

Figure III.2.a shows an overview of the processes comprising the MPI universe and their connection to the underlying POETS hardware. MPI was designed in an era when most multi-process systems ran on supercomputers and were highly homogeneous. The interprocess communication choreography was designed by the software architect, and was usually regular, predictable and tightly controlled. The Orchestrator takes MPI away from this design intent, and as you might expect, if you use a tool for a purpose for which it was not originally intended, strange things can happen.

Interprocess communication in the Orchestrator universe is asynchronous, unpredictable, heterogeneous and is encapsulated in a small subsystem: the **`MPISpinner`** in **`CommonBase`**. ***All*** MPI interprocess messages *without exception* are wrapped in a POETS object called **`PMsg_p`**, which is itself a thin derivation from a generic called **`Msg_p`**. ***All*** sends without exception are launched from **`PMsg_p::Send`** (or occasionally **`PMsg_p::Bcast()`**); ***all*** receives without exception land in **`CommonBase::MPISpinner()`**. **`Msg_p`** is a generic container whose contents are arbitrary, dynamic and may be accessed randomly. Think of it as a dynamic structure.

2.1 CommonBase and system startup

All Orchestrator process principal classes are derived from **`CommonBase`**. Referring to **`CommonBase::CommonBase`**, the system starts MPI in hybrid mode (some of the processes are multi-threaded: this has found to be a stable startup protocol). It then *replaces* the MPI default immediate send message buffer with a local buffer of 1Gbyte *on every process*. This seems to be the largest possible in a 32-bit system, although no maximum size is documented. Test commands allow even this buffer to be overwhelmed, but the circumstances are highly artificial. (Thus the system outlined in figure III.2.a has a total universe footprint of many Gbyte on startup.)

Each process contributes an element to the **Process Map** (figure III.2.1.a). This is simply a structure containing the MPI rank, machine name, creation details and a few other details for each process that is broadcast to the universe (wrapped as a **`PMsg_p`**), so everyone knows where everyone else is and how they may be found.

It then enters an MPI barrier (**`CommnBase::CommonBase - Prologue()`**), to allow all the POETS processes to assemble their copy of the process map for the entire universe. Once through the barrier, control passes back up through the constructor hierarchy (figure III.4.1.a) and starts the MPI message spinner.

2.2 The MPI spinner

As the name implies, all the processes in the universe spin on an MPI `Improbe,` waiting for a message to which they must react. The structure of `CommonBase::MPISpinner` is shown in figure III.2.2.a.

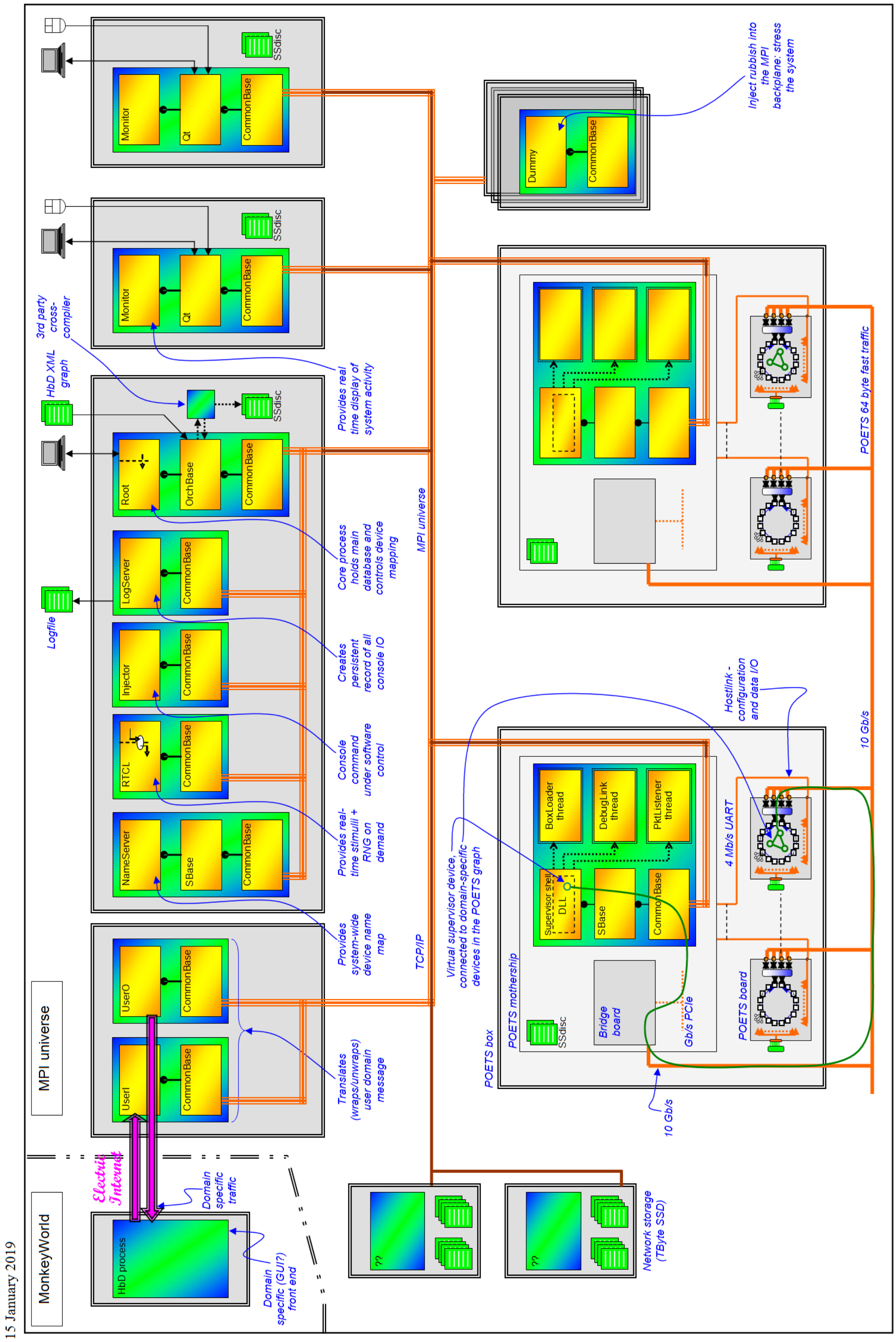

15 January 2019
MonkeyWorld
MPI universe
Electric Internet
Domain specific traffic
HbD process
Domain specific (GUI?) front end
UserI
CommonBase
UserO
CommonBase
Translates (wraps/unwraps) user domain message
NameServer
SBase
CommonBase
Provides system-wide device name map
RTCL
CommonBase
Provides real-time stimulii + RNG on demand
Injector
CommonBase
Console command under software control
LogServer
CommonBase
Creates persistent record of all console IO
Logfile
Root
OrchBase
CommonBase
Core process holds main database and controls device mapping
HbD XML graph
3rd party cross-compiler
SSdisc
Monitor
Qt
CommonBase
SSdisc
Provides real time display of system activity
Monitor
Qt
CommonBase
SSdisc
MPI universe
TCP/IP
Dummy
CommonBase
Inject rubbish into the MPI backplane: stress the system
Virtual supervisor device, connected to domain-specific devices in the POETS graph
POETS box
POETS mothership
BoxLoader thread
DebugLink thread
PktListener thread
Supervisor shell DLL
SBase
CommonBase
SSdisc
Bridge board
Gb/s PCIe
10 Gb/s
4 Mb/s UART
POETS board
Hostlink - configuration and data I/O
10 Gb/s
POETS 64 byte fast traffic
??
??
Network storage (TByte SSD)

Figure III.2.a: Orchestrator coarse structure

```
Class RTCL rank 5 compiled with 32-bit Borland compiler 560 under Windows
from E:\Grants\POETS\Software\ZeOrchestratings\Source\RTCL\RTCL.cpp
to   E:\Grants\POETS\Software\ZeOrchestratings\Borland\RTCL1.exe
at   10:48:29 on Aug 30 2017
executing on isdfc.ecs.soton.ac.uk as user ADB

MPI hybrid programming model: Thread-serialized
MPI asserts time is global
MPI timer tick 4.277308e-07 seconds
- - - - - - - - - - - - - - - - - -
Rank 4 asserts: rank 0 has IO
```

Figure III.2.1.a: Process map element

2.3 Message decode

The **`CommonBase`** constructor builds the process map (previous section) and the **message map**.

2.3.1 POETS <u>messages</u>

The POETS Orchestrator is a multi-process system, communicating via the MPI message-passing standard. Although MPI permits a rich variety of message structures, to simplify the intercommunication choreography, **all** messages sent over the MPI network are of the form - from the perspective of MPI - of a simple byte vector. This vector naturally has internal structure, but this is not exposed to the MPI system. **Every** POETS message is a streamed instance of class **`Msg_p`**[1]. The term **message** refers (in the context of POETS) to a streamed **`Msg_p`** object, not a generic MPI message, unless explicitly stated otherwise.

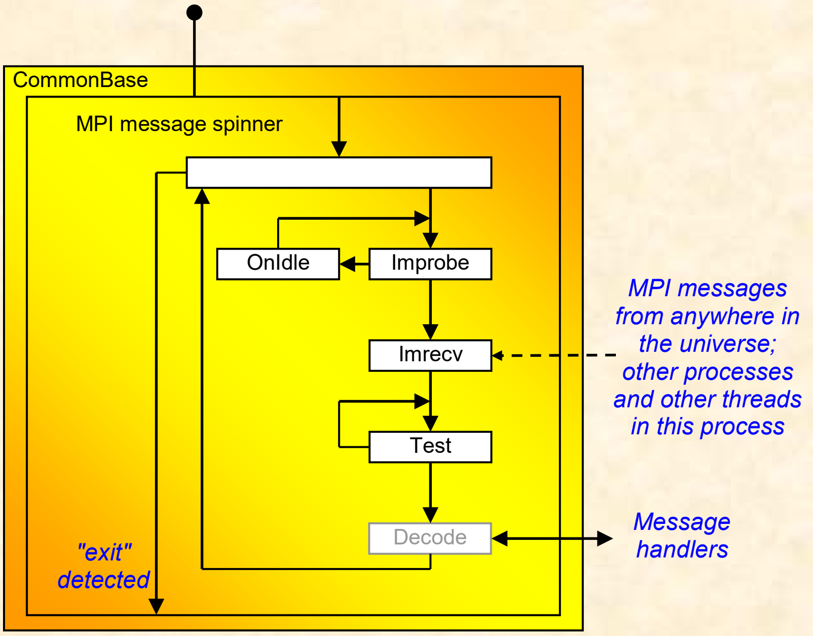

CommonBase
MPI message spinner
OnIdle
Improbe
Imrecv
MPI messages from anywhere in the universe; other processes and other threads in this process
Test
Decode
Message handlers
"exit" detected

Figure III.2.2.a: MPI spinner

[1] **`Msg_p`** is a generic, documented separately. It is one of the more complicated generics, and if necessary the reader is strongly advised to branch to that documentation here before continuing.

In precis, a **Msg_p** object is a container. It holds *variable* fields (the **Msg_p** *payload*) and *constant* fields.

Msg_p variable fields:

A **Msg_p** object can hold an arbitrary number of typed data elements. Each data element is paired with an arbitrary integer ***key***:

Putting:

```
Msg_p X;
double d=9.9;
X.Put<double>(6,&d);
```

Getting:

```
double *pdbl = X.Get<double>(6,cnt);
if (pdbl!=0) d = *pdbl;
```

Data can be PoD, any user-defined structure that does not have dynamic components, and even some special STL cases that do have dynamic components (these are treated as special cases within **Msg_p** - strings, vectors - but they are so useful it is worth it).

As defined, **Msg_p** (a generic) has no MPI dependences, and can be used as any other kind of container in a single process program. **PMsg_p** is derived from **Msg_p** and encapsulates the MPI dependencies.

Aside from its value as a generic tuple-like container, **Msg_p** has the ability to stream itself into (and reconstruct itself from) a self-unpacking byte stream. In the context of a single thread program, there is little value in this - you might as well pass around the **Msg_p** object and let the compiler do the work - but this aspect is invaluable in a(n MPI) multi-process system.

A derived class, `PMsg_p`, has a dependency on MPI, and some extra methods:

- **`PMsg_p.Send(int rank)`** causes the (arbitrarily complicated) **Msg_p** to pack itself into a byte vector and despatch itself to the target rank process.

- **`PMsg_p.Send()`** as **`Send(int)`**, but the target is taken from the internal message field **`Msg_p::tgt`**.

- **`PMsg_p.Bcast()`** sends the message to the entire MPI universe. This is not a wrapper for **`MPI::Ibcast`**, because reliability issues were found in stress testing. It is just a loop through the universe processes, one at a time. Slow, but in the Orchestrator use case it doesn't matter.

At the target process, the container can unpack itself:

```
.
MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&flag,&status);
MPI_Get_count(&status,MPI_CHAR,&count);
MPI_Irecv(&MSGBUF[0],count,MPI_CHAR,MPI_ANY_SOURCE,
          MPI_ANY_TAG,MPI_COMM_WORLD,&request);
.
PMsg_p Pkt((byte *)&MSGBUF[0],count);
```

.

Msg_p constant fields:

The Msg_p object contains a small number of fixed housekeeping fields:

```
int tag,id,mode,src,tgt
double ztime[4]
string zlabel[4]
byte subkey[4]
```

The field names reveal their design origins, but in the context of POETS most can currently be used for anything. (MPI represents time as a `double`; **`ztime`** was used to measure transit timings. **`Subkey`** *is* used by POETS - see next section; I can't remember what the point of **`zlabel`** was.)

By using **`PMsg_p`** objects, and *only* **`PMsg_p`** objects, the entire Orchestrator MPI universe can be made asynchronous. Nothing knows what to expect or when, but it doesn't matter because (from the MPI perspective) all messages are the same (have the same MPI type).

2.3.2 Decoding messages

Each incoming message has four single byte **`subkey`** fields built into it - see figure III.2.3.2.a. These are `unsigned chars`, and trivially concatenated into an **`unsigned`** key value. Each principle class has two message maps: one in **`CommonBase`** and one in the derived class. The message map (**`FnMap`**) is a **`std::map`** container object, that maps an `unsigned` key to a method function address within the object:

```
typedef unsigned       (CommonBase::*pMeth)(PMsg_p *);
map<unsigned,pMeth>    FnMap;
```

This allows messages to be sent to the correct handler function simply by dereferencing through the appropriate message map - see figure III.2.3.2.b.

At the heart of the procedure is the message spinner (figure III.2.2.a). This spins on **`MPI_Improbe()`** (the multi-thread version of **`MPI_Iprobe()`**) in **`CommonBase::MPIspinner.`**

`MPI_Imrecv()` collects messages from anywhere in the Universe: These can be commands sent from the keyboard thread in the **`Root`** process, or messages from any other process. A second tight loop on **`MPI_Test()`** waits for the message to arrive (Micro$oft **`MPI_wait`** does not appear reliable in a hybrid process).

The message is then handed into **`CommonBase::Decode`**, (figure 2.3.2.b) which is pure virtual, and so redirects to **`Root::Decode`**. This offers the message up to **`Root::FnMap`**. If **`Root::FnMap`** has an entry for that message (that is, the key fields match), the appropriate **`Root`** function is called. If **`Root::FnMap`** has no entry, the call drops back to **`CommonBase::FnMap`** (**`FnMap`** is *not* virtual - the redirection back down the inheritance hierarchy is explicit). If the message is not handled here, what information is available is sent to the LogServer as an error. (Without a handler, only the key and source rank will make sense in a message, because the receiver (**`Root`** or **`CommonBase`**) by definition have no knowledge of the internal structure of the message.) Ultimately, this will be a "should never happen" message.

2.4 CommonBase message handlers

These handlers - as one might expect - handle messages that may legitimately arrive at any POETS process.

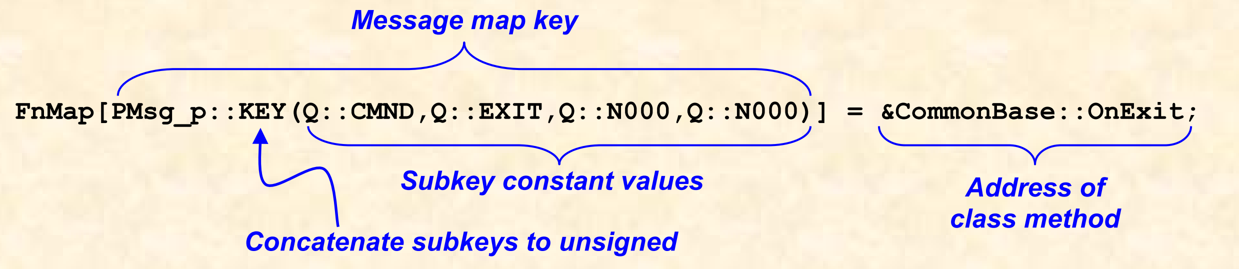

Message map key
FnMap[PMsg_p::KEY(Q::CMND,Q::EXIT,Q::N000,Q::N000)] = &CommonBase::OnExit;
Subkey constant values
Address of class method
Concatenate subkeys to unsigned

Figure III.2.3.2.a: Message map element

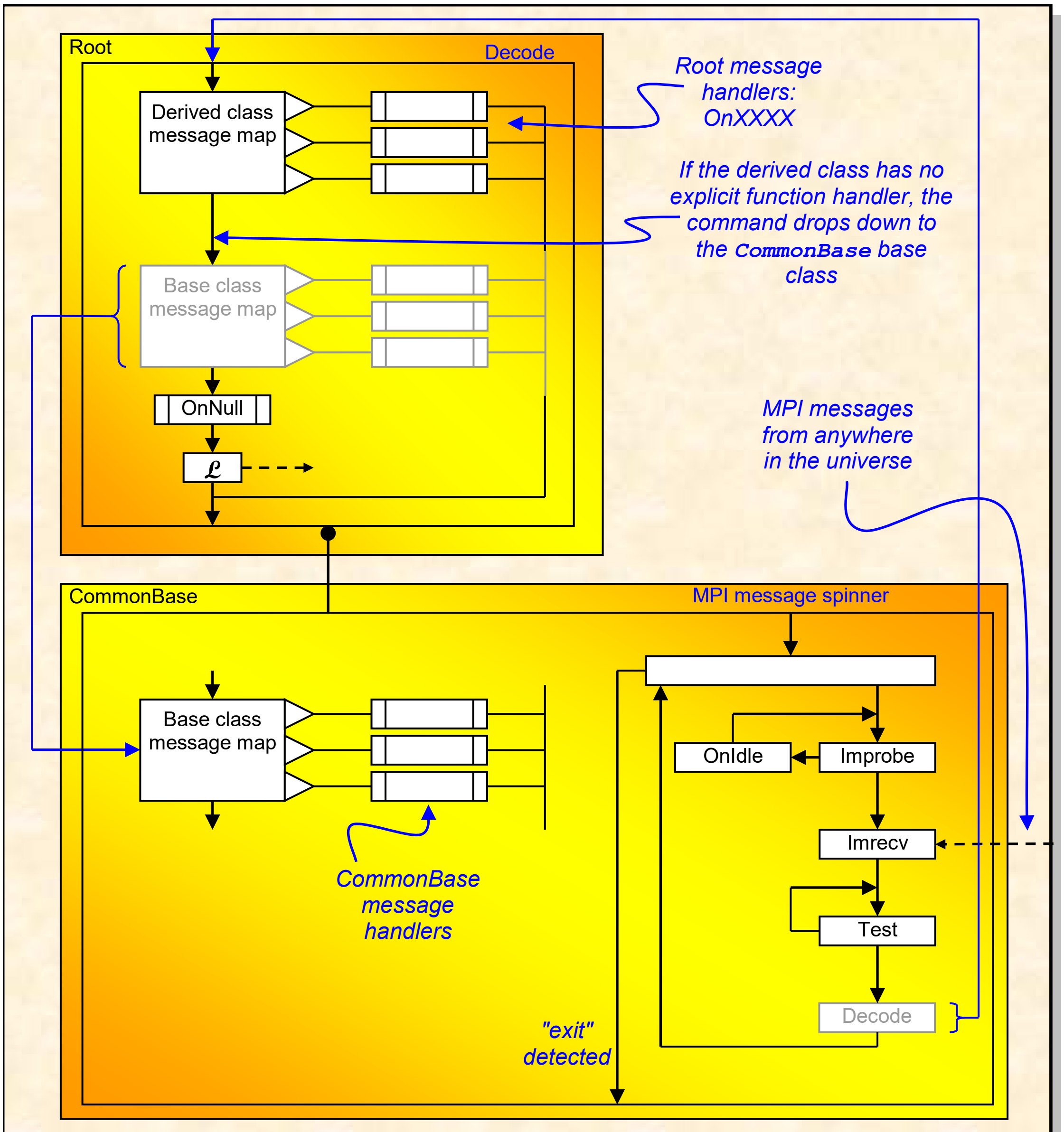

Root
Decode
Derived class message map
Root message handlers: OnXXXX
If the derived class has no explicit function handler, the command drops down to the CommonBase base class
Base class message map
OnNull
£
MPI messages from anywhere in the universe
CommonBase
MPI message spinner
Base class message map
OnIdle
Improbe
Imrecv
Test
Decode
CommonBase message handlers
"exit" detected

Figure III.2.3.2.b: Decoding messages

OnIdle:
This is called in the inner loop of the message spinner (figure 2.2.a). It obviously blocks the loop, so must be fast. The base class functionality is empty; it is virtual and may be overridden by the derived class. This is currently done in the Root process to handle batch commands - see section III.3.6. Uniquely amongst the message handlers the return type is void.

OnPing:
This handles ping requests usually (but not essentially) emanating from the Root. The ping REQ message is timestamped, turned into an ACK and returned to sender.

OnPmap:
This handler is used to add an element to the process map on startup discussed earlier. (Note these can arrive in any order.)

2.5 System shutdown

Each process closes down on receipt of an "exit" message. This is broadcast from the Root when the user initiates closedown; the only subtlety is the order in which the messages are sent: Root broadcasts the closedown message to every process *except* the `LogServer` (because the `LogServer` has to be active to acknowledge closedown back to the console and write to and close the log file), then the `LogServer`, and finally itself. Each multi-threaded process obviously has to close down all its own threads.

Exits can also be scheduled by the operator (see the user guide). The trigger for these exits is checked in `OnIdle`, which is only performed when the message queue is empty.

OnExit:
All the message handlers (except **`OnIdle`** - see figure III.2.2.a) return an `unsigned`, which is used to indicate closedown if !=0. (This is how the "exit" command propagates through the code.) **`CommonBase::OnExit()`** does nothing but return 1 to force closedown of whatever process it is in.

2.6 CommonBase::Dump()

A typical **CommonBase** datastructure dump is something similar to figure III.2.6.a. The full process map will contain one element (see figure III.2.1.a) for each process in the Universe. The unsigned key fields in the event (message) map are enumerated in the file **Pglobals.cpp|.h**; the actual method addresses are compiler generated and only useful if you're doing some desperate debugging.

```
CommonBase++++++++++++++++++++++++++++++++++++
Sderived (this derived process) : Root
Event handler table:
Key        Method
0x02010000 0x0041d83c
0x08000000 0x0041ee5c
0x09408000 0x0041d8d4
0x09408100 0x0041d8d4
S00 (const static)              :
Urank                           : 0
Usize                           : 6
Ulen                            : 21
Sproc                           : isdfc.ecs.soton.ac.uk
UBPW                            : 32
Scompiler                       : Borland compiler 560
SOS                             : Windows
Ssource                         :
E:\Grants\POETS\Software\ZeOrchestratings\Source\Root\Root.cpp
Sbinary                         :
E:\Grants\POETS\Software\ZeOrchestratings\Borland\orchestrator010.exe
STIME                           : 15:02:08
SDATE                           : Aug 30 2017
Process Map start
.
.
.
```

Figure III.2.6.a: CommonBase::Dump pretty print

3 Root process

`Root` is derived from two base classes, **`CommonBase`** (section III.2) that handles MPI message brokering, and is common over the Universe, and **`OrchBase`** (section III.4), that handles the main business of the Orchestrator: building the processor topology graph and the device (application) graphs, cross-linking the two and building the binaries to load into the POETS cores.

3.1 Function and responsibilities

The principal Orchestrator process is **`Root`**: this oversees all the heavy processing (it is more complex than all the other processes put together) and it also interacts with the user. (In MPI-speak, it has rank 0 and owns the IO.) The MPI Universe is asynchronous, in the sense that no process is ever actually *expecting* any messages. Every process 'main' class is derived from **`CommonBase,`** that contains a spinner - outlined previously - continually probing its MPI queue to see if there is a message for it. This is at odds with any kind of sensible keyboard response, so **`Root`** spins off a lightweight thread that *itself* spins on a blocking keyboard read from the user console (ANSI C++ has no non-blocking keyboard read), wraps user commands into a message, and sends this message to its own main thread - effectively the *process* is sending a message to itself across threads. (Hence the 'hybrid' MPI dialect startup call.) See figure III.3.1.a.

All processes in the Universe have message handlers, that react to incoming MPI messages. By convention, these are of the form **`OnXxxx`**, and have a fixed signature so their address can be placed in the function handler method map. The **`Root`** process has a thread that reads strings from the console. These have a limited (legitimate) structure, such that they can be interpreted by the Command Line Interpreter (**`Cli`**) **generic** class. Once interpreted, the individual commands are supported by a set of methods of the form **`CmXxxx`**. The various

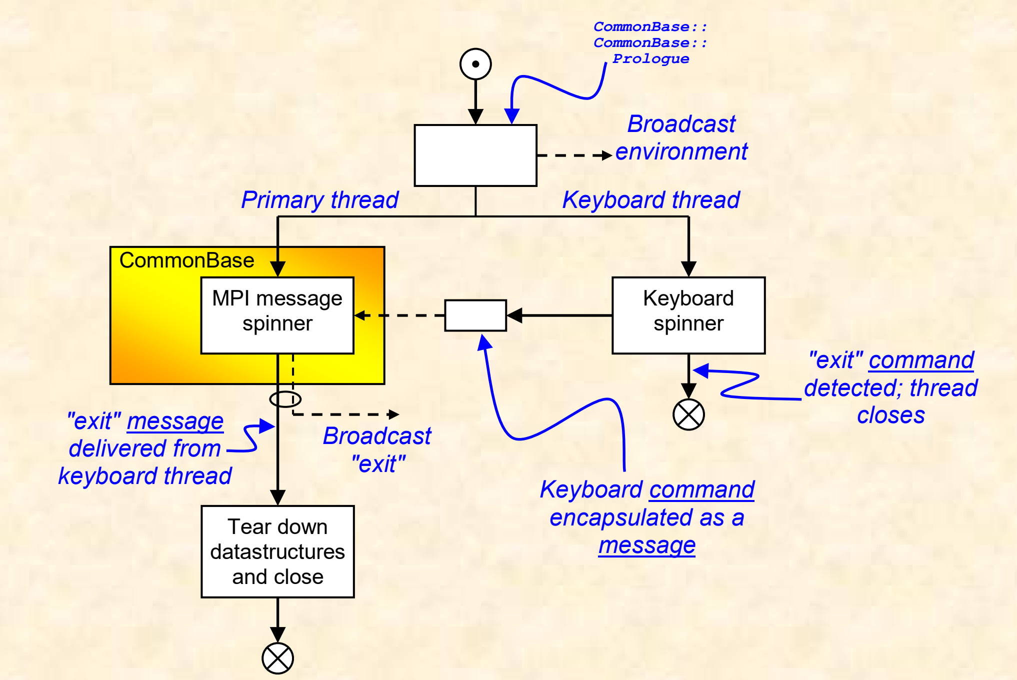

CommonBase::
CommonBase::
Prologue
Broadcast environment
Primary thread
Keyboard thread
CommonBase
MPI message spinner
Keyboard spinner
"exit" command detected; thread closes
"exit" message delivered from keyboard thread
Broadcast "exit"
Keyboard command encapsulated as a message
Tear down datastructures and close

Figure III.3.1.a: Root internal layout

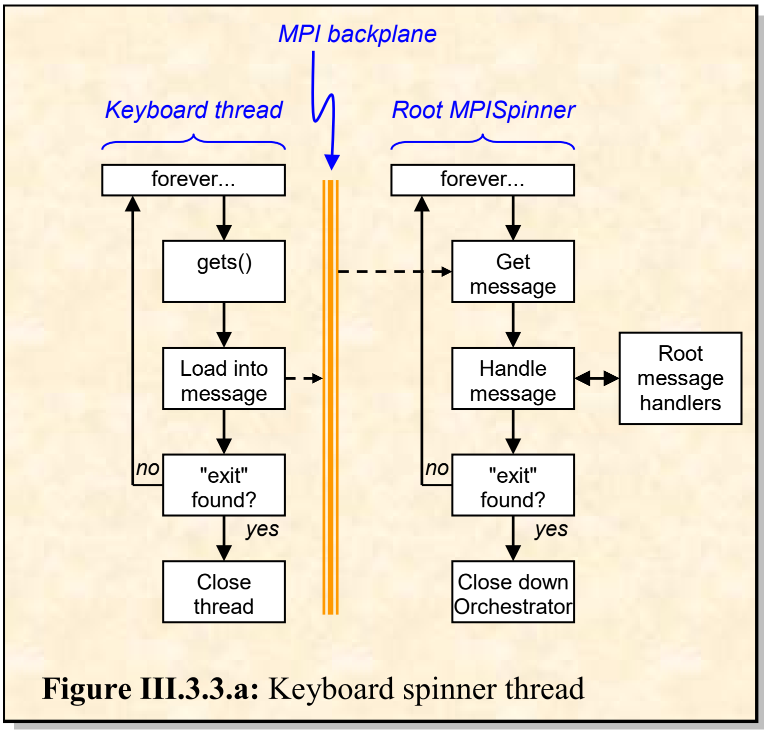


Figure III.3.3.a: Keyboard spinner thread

subsidiary methods are less structured.

3.2 Internal structure

There is no large structure here to speak of - see figure III.3.1.a - most of the complexity lies in the base class `OrchBase`. Root hosts

- Part of the datastructure for the injector subsystem (it really has to span two processes, and however you distribute it one side looks uncool - see section III.9.)
- The batch file subsystem (section III.3.6) is non-trivial, because it has to be able to interleave batch commands with incoming asynchronous messages from the rest of the Universe.
- The command processor - this turns console commands into MPI messages and despatches them to the appropriate process.

3.3 Keyboard thread

The internal control flow of the keyboard spinner is shown in figure III.3.3.a. `Root::Root` spins off a thread entered by `kb_func()` defined in the compilation unit Root.cpp.

The thread reads the keyboard with `fgets(...,stdin)`, as opposed to `gets(...)`. The former requires the up-front definition of the buffer length, and will not overflow it; the latter allows the user to overflow the buffer by providing an over-long input record - generally considered a bad thing.

`fgets()` terminates input *either* when `'\n'` is detected *or* the buffer is full - hence the rather convoluted code (compounded by a Borland compiler bug[2]) to ensure the character array is

[2] www.delphigroups.info/3/4c/27471.html

terminated by a `'\0'` under all circumstances. Note that the buffer is a C array of char; this is not the same thing as a C char * or C++ std::string. `'\0'` has no special meaning until it is interpreted by a **`<stdio> strxxx`** function or **`std::string`** constructor.

3.4 Command line interpreter

The command line interpreter class (Cli) is a **generic** class documented separately. In precis, it deconstructs strings of the form

```
command [/clause [=[op]param[,[op]param]]]
```

and exposes the various constituents in a useful and convenient manner. It has a vestigial API; the datastructure is intended and designed to be burgled.

Parameters may be a **simple** name (string) or a **compound** name of the form `[op]param[::param]`^{n}.

In order to facilitate string comparisons (when you're trying to decode what the user has typed), a method is provided that allows command and clause (but not parameter) strings to be truncated to a fixed length. The Command Portfolio (section 4.10) describes this further: commands and clauses are truncated to **four** characters.

3.5 Message handlers

These handlers look after messages that are only applicable to the `Root` process:

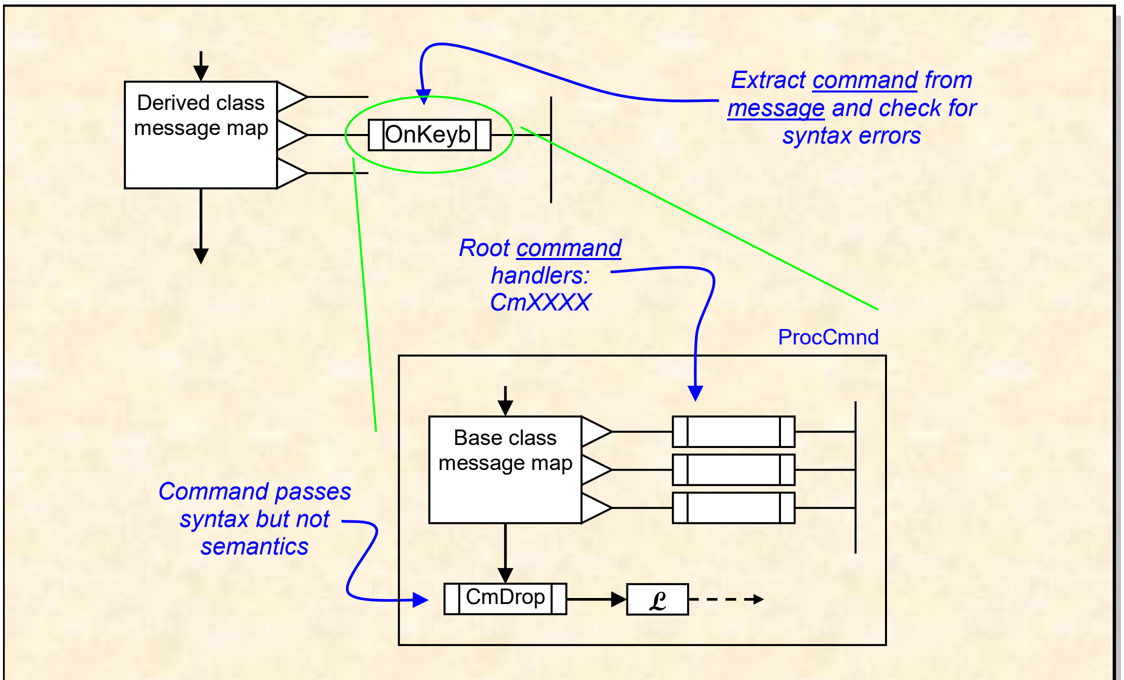

Derived class message map
OnKeyb
Extract command from message and check for syntax errors
Root command handlers: CmXXXX
ProcCmnd
Base class message map
Command passes syntax but not semantics
CmDrop
ℒ

Figure III.3.5.a: Decoding commands from messages

OnKeyb:
This message handler unpacks the command string from its message and checks the syntax. This is a very low overhead operation, and so can be carried out many times if necessary - i.e. it is usually more convenient to pass around the **`Cli`** object by value than its various component parts. Here we check the syntax (not semantic content), and reject the command if necessary. If it is syntactically correct, it is passed into yet another decoder (**`Root::ProcCmnd`**) and forwarded to a *command handler* (like a message handler, only different) - see figure III.3.5.a.

OnLogP:
This message handler receives an elaborated message (see section III.7) from the **`LogServer`** and writes it to the user console.

OnTest:
Message handler for testing and breaking; it is not intended to be generally visible - or the contents stable.

3.6 Batch command handling

The Orchestrator system provides low-level and close control over what is going on internally, and it requires multiple commands to achieve anything useful, although this is intended to change as the system becomes more mature. A capability to transfer control temporarily to a batch file is of obvious utility. However, this is not just a question of

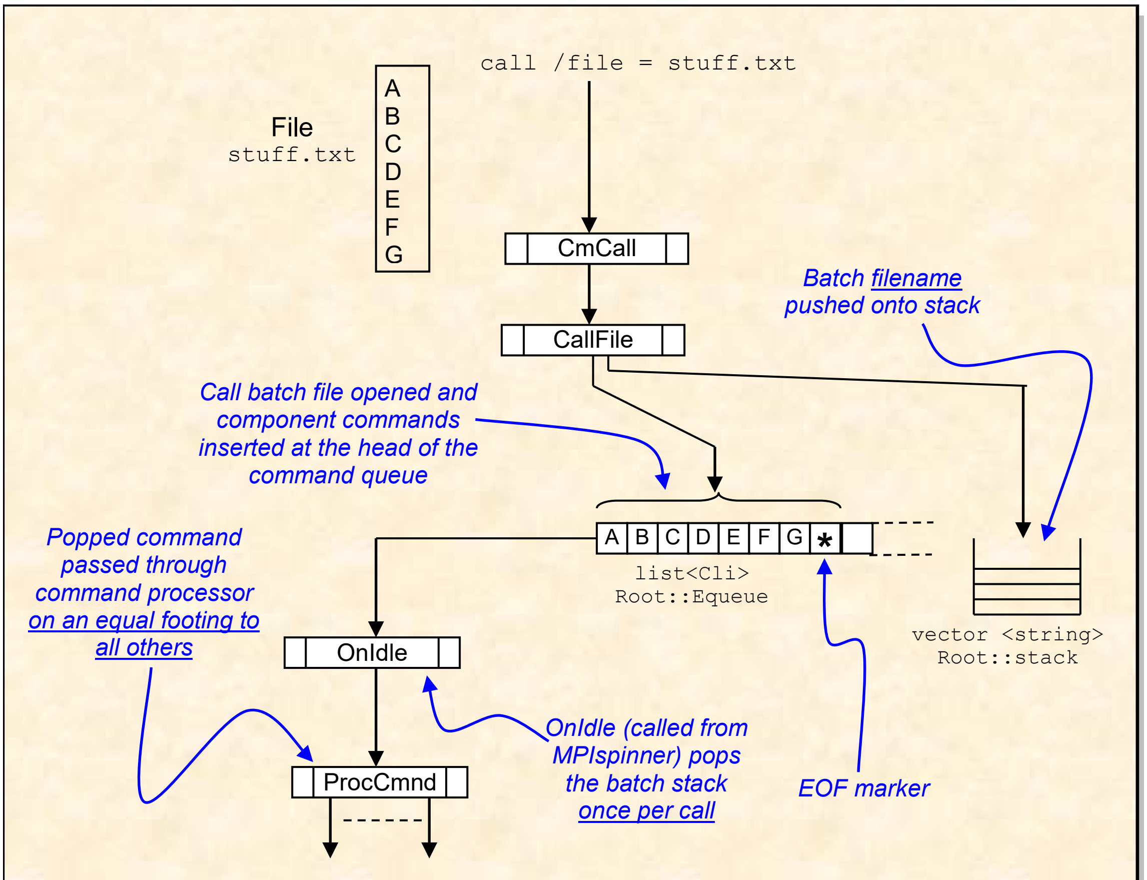

call /file = stuff.txt
File
stuff.txt
A
B
C
D
E
F
G
CmCall
CallFile
Batch filename pushed onto stack
Call batch file opened and component commands inserted at the head of the command queue
Popped command passed through command processor on an equal footing to all others
A B C D E F G *
list<Cli>
Root::Equeue
vector <string>
Root::stack
OnIdle
ProcCmnd
OnIdle (called from MPIspinner) pops the batch stack once per call
EOF marker

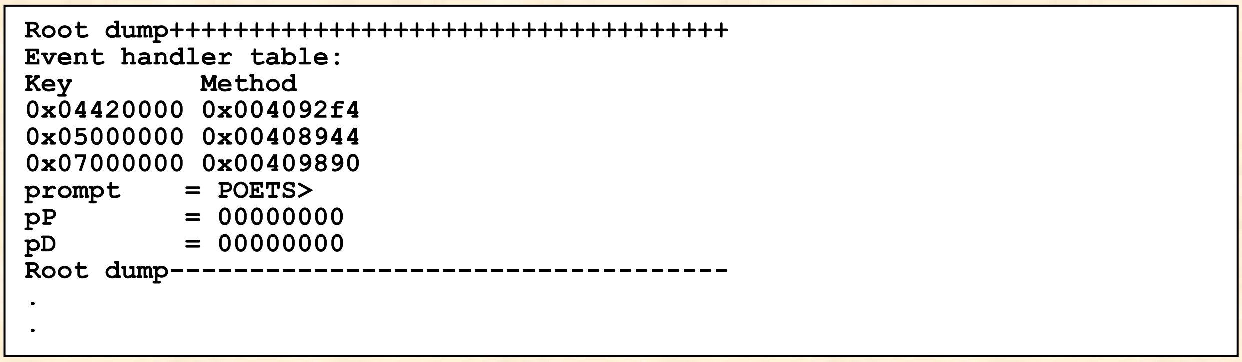
Figure III.3.6.a: Executing batch commands

switching the input stream to a file, as in a single-thread system. All console writes go (in compressed form) via the **`LogServer`** (section III.7), so it can archive a timestamped copy. The elaborated message for the console itself is posted back to the **`Root`**, fielded by **`Root::OnLogP`** (section III.3.5) and written to the screen. If input stream control for multiple commands is handed to a batch file, the command responses will not be interleaved correctly with the commands, because the Root MPI Spinner will not get control until the entire batch file has executed.

To get around this, every batch file is loaded into a double ended queue of commands, and the **`Root::OnIdle`** handler overload pops exactly one command from the queue head every time **`CommonBase::OnIdle`** is executed (figure III.3.6.a).

The commands in **`stuff.txt`** can be any legal Orchestrator console commands (which includes further **`call`** commands). These are handled by the scheme outlined above: each new invoked batch file is unpacked and pushed into the *front* of the command queue. Alongside this, every time a batch file is loaded into the command queue (**`list<Cli> Root::Equeue`**) the *name* of the file is pushed into a call stack (**`vector<string> Root::Stack`**). The filename stack has no other purpose than to prevent recursive batch file execution: if a call command is encountered (anywhere, but meaningfully from within a batch file), the putative file name is checked against the contents of the stack; if present, the call command is ignored.

The end of file command inserted into the queue does nothing external on execution; it indicates that the filename stack should be popped.



```
Root dump+++++++++++++++++++++++++++++++++++
Event handler table:
Key        Method
0x04420000 0x004092f4
0x05000000 0x00408944
0x07000000 0x00409890
prompt    = POETS>
pP        = 00000000
pD        = 00000000
Root dump-----------------------------------
.
.
```

Figure III.3.7.a: `Root::Dump` pretty print

3.7 Root::Dump()

Figure III.3.7.a shows a Root datastructure pretty-print. Apart from the message map, it contains only the console prompt string and some pointers into the **`OrchBase`** object.

4 The OrchBase class

This class contains by far the largest and most complex component of the entire system. It contains a model of the hardware (as is necessary to perform its function), and models of all the application tasks. It performs the mapping of the latter to the former, builds all the necessary datastructures (including the binary files necessary to drive the low-level P-cores), oversees their transport to and loading into the hardware, and the system execution.

4.1 Class derivation hierarchy

The class hierarchy and initialisation control flow for the main Orchestrator process, Root, is shown in figure III.4.1.a.

This section describes the various inheritance hierarchies present in the Orchestrator.

All Orchestrator processes are implemented as a dynamic objects, *all* of which are derived from **`CommonBase`**, which supports all inter-process communication using MPI. A spinner is located in **`CommonBase`** to support asynchronous communications. This is described in detail in section III.2.

The main Orchestrator process, Root, is a three-stage inheritance:

```
{CommonBase,NameBase}->OrchBase->Root
```

The (original) idea was to have the datastructure in **`OrchBase`** and the command processors in **`Root`**, but over time this morphed into the message handlers in **`Root`** and the command processors in **`OrchBase`** alongside the datastructure they act upon.

Most of the objects in the datastructure are derived from some subset of three base classes: **`DefRef`**, **`DumpChan`** and **`NameBase`** (see figure III.4.1.b). All of these are generics and more fully described in their stand-alone documentation, but for the sake of convenience a precis is:

`DefRef`: Provides definition/reference counting for derived higher-level objects. It assumes that every definition and every reference is defined by a line number in some file defined elsewhere. The internal state has two items: a definition (**`unsigned`**, initialised to 0 that means *undefined*) and a **`vector<unsigned>`**, that contains the lines that reference the item. The internal state is private, and the API allows the inherited class to do all sensible manipulations. *Setting/resetting the object is not done automatically; the Def/Ref functionality does not refer to the C++ level of referencing, rather to some user-defined higher level.*

`DumpChan`: This is an adjunct class to the generic **`pdigraph`** template. (It's been a subclass, then not, and so on over the years...). **`Pdigraph`** is a template that holds tripartite **`{node, edge, connector}`** graphs. To assist introspection, **`pdigraph`** contains routines that will walk its internal structure, executing a (user-defined) callback on each of the three elements. Typically this will be some sort of Dump function, which needs a valid output stream. In order for the output to make any sort of sense, invocations of the callbacks are interspersed with chaperone writes from **`pdigraph`** itself. It follows from all this that the objects in the template (which are a priori unknown to the template) and the template itself must have access to a common output stream. Numerous ways of doing this exist; here we derive **`pdigraph`** from **`DumpChan`**; the sole state member of **`DumpChan`** is a **`static FILE *`**. Externally deriving the classes placed into the template from **`DumpChan`** allows everything to access a common output stream.

`NameBase`: Allows a group of objects to form a name hierarchy. Simplistically, every **`NameBase`** instantiation has a *local name* and a parent **`NameBase`**. Reading/writing a local name is as you would expect, but it is also possible to call **`string FullName()`** on an instance of the derived class and retrieve the full hierarchical name. Formatting methods allow control over the level of detail returned by **`FullName`**.

The only other class derivation occurs in the data structure where the supervisor type definition class **`SupT_t`** is derived from the (non-supervisor) device type class **`DevT_t`**.

4.2 Code layout

The Root process code layout is shown in figure III.4.1.a.

- The passage of control through the constructor and class hierarchy is shown by the thick red line.
- Dotted red lines mean control/create/interact with.
- The thick orange line is the MPI backplane.

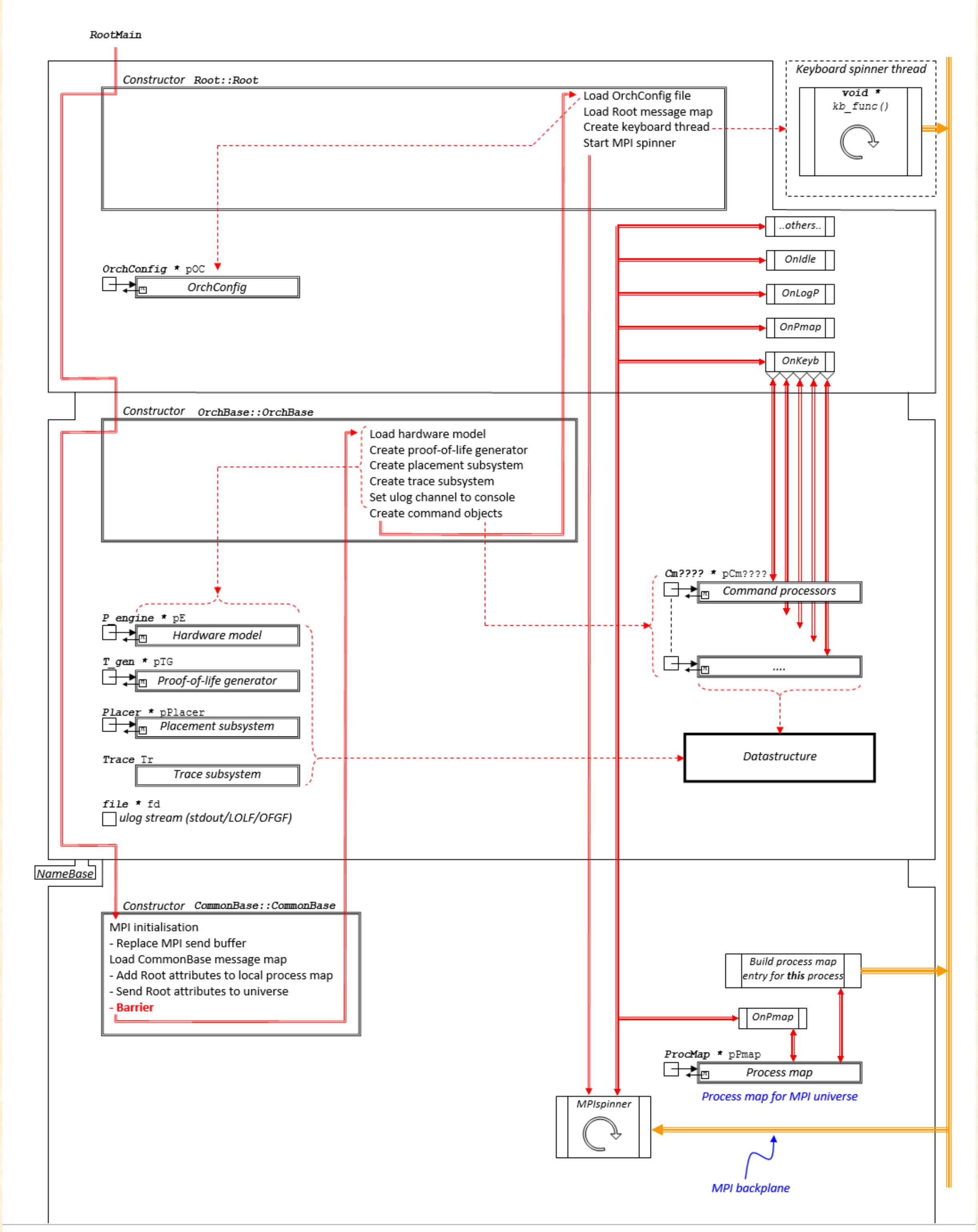


Figure III.4.1.a: Root process code layout

RootMain is executed by the Launcher (section III 14.2), and control passes through the object hierarchy constructors to **CommonBase**. Main (not all) points of relevance here are

- The MPI send buffer is replaced by a much larger-than-default buffer in the **CommonBase** constructor. This was added as an outcome of stress-testing the communications infrastructure (message storms can overwhelm the MPI buffer). It could probably be replaced with a smaller one without detriment in normal use, but on the other hand, it exists in a 32-bit space. Notwithstanding the comments in the introduction, moving some or all of the system to a 64-bit platform is a work-in-progress.
  Note that the MPI standard suggests (but does not mandate) that **malloc/free** is used, rather than **new/delete** for manipulating the message buffer.

- **CommonBase** then accumulates environment parameters for itself, and broadcasts these (the Process map entry) to the MPI universe. As the MPI spinners in every Orchestrator spinner are asynchronous, these can arrive anywhere in any order, so every Orchestrator process goes into an MPI barrier until everyone has an accurate image of the process map for the entire universe.

- Not shown in figure III.4.1.a is the exit code layout. The MPI spinner has a closedown exit path that is not trivial. Closedown is managed by **OrchBase::CmExit**; in essence, the various processes need to be sent an "exit" command, and the order of events after that (controlled by **CmExit**) is {closedown acks from everyone except LogServer, closedown ack from LogServer, closedown Root}.

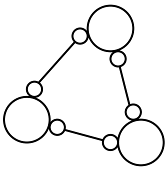
- The command processors correspond pretty much one-to-one with the user commands, and are instantiated by the **CommonBase** constructor. However, the instantiation is not complete at this stage. For example, the **load** command handler requires that incoming XML be parsed and validated, and for this the configuration files are needed. However, at the point at which the **load** command handler is created (in the **CommonBase** constructor) the **Root** constructor body has yet to be invoked, so the location (and validity) of the configuration files is unknown. The complete instantiation of the **load** handler is thus done 'on demand'. When/if the user executes a **load**, if the necessary objects have not (yet) been created, they are instantiated at that point. This is only done once, but it is at this point that the configuration files are checked for validity, not on startup, which can confuse a user if errors are detected.

- Each object generally has a parent backpointer (**π**) to its 'owning' object making two-way traversal of the structures easy.

- ***Teardown***: disregarding the crosspointers (section III.4.3.4) the entire structure is a pair of trees: platform and application. Calling delete on an application (**Apps_t** object) will delete its child **GraphI_t** and **GraphT_t** objects, and so on down the trees. The pdigraphs all contain *pointers* to objects; when the enclosing object is deleted, it must walk the container, explicitly deleting the contents, before deleting the container (the pdigraph) itself.

4.3 Datastructure

Central to the datastructure are multiple instances of the generic template class **`pdigraph`**. Like **`Msg_p`**, pdigraph is a complicated class, and if unfamiliar with it a branch is recommended here. **`Pdigraph`** represents a directed ***trigraph*** {vertices, edges, connectors}. (A connector must be associated with each end of an edge.)

Pdigraph in figure III.4.1.b: 

The Orchestrator datastructure is shown in figure III.4.1.b. It is dense (an A3 version is available).

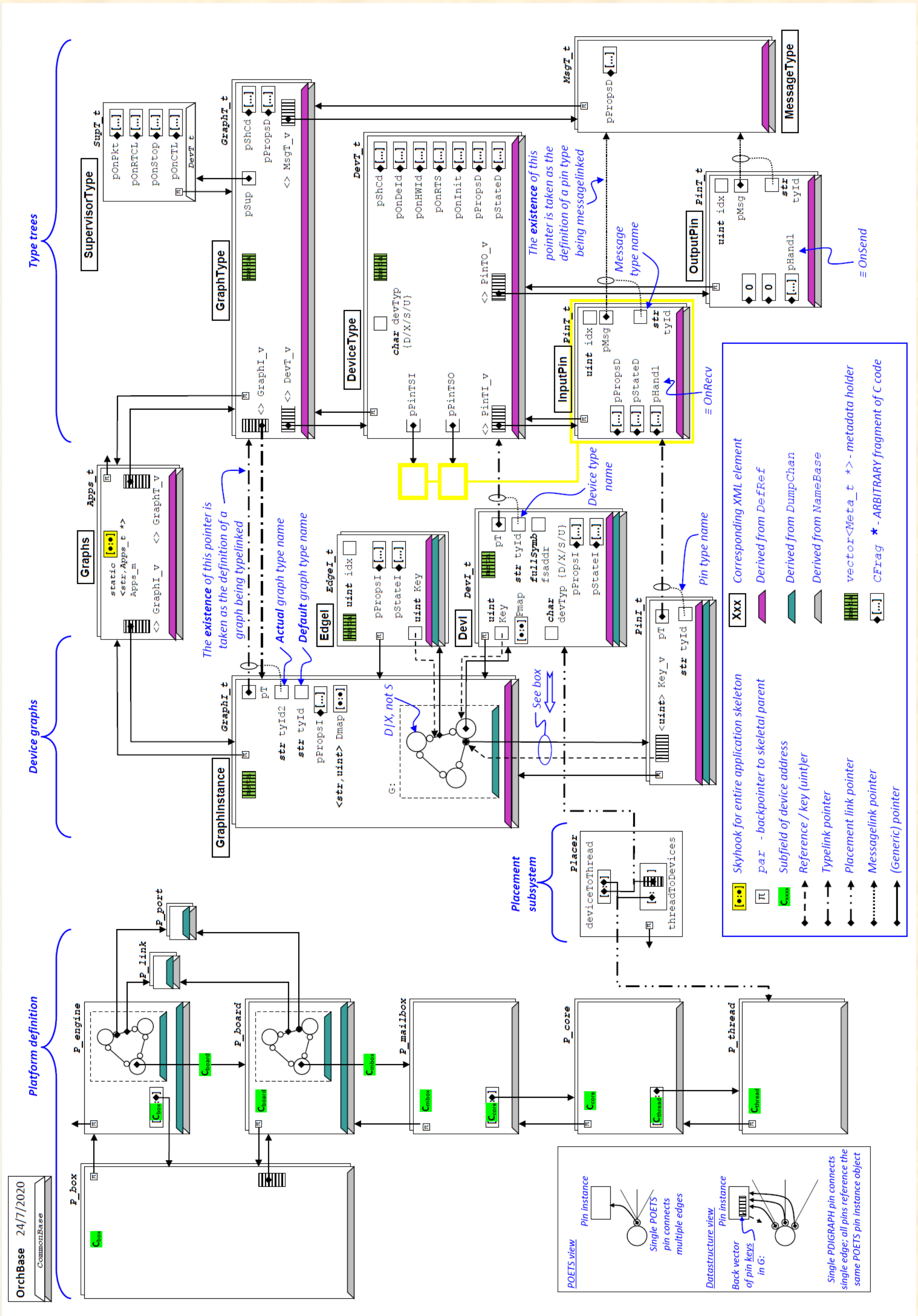


Figure III.4.1.b: OrchBase data structure

4.3.1 Philosophy

The principal responsibilities of the Orchestrator are to facilitate the loading and execution of (multiple) user-defined **applications**. Before describing the datastructure in detail, it is helpful to review what it is designed to support.

The application graph is defined as a flat device graph, described previously. At least one application graph lies at the heart of every user application.

The hardware is a complex hierarchy of computing threads, which is reconfigurable (when the parent FPGAs are programmed). This hierarchy is - loosely - a hardware graph of processing capability.

The principal role of the Orchestrator is to take a set of applications, map them onto the hardware, create the necessary binaries and download them to the hardware.

Figure III.4.3.1.a shows a simplified illustration of the Orchestrator behaviour.

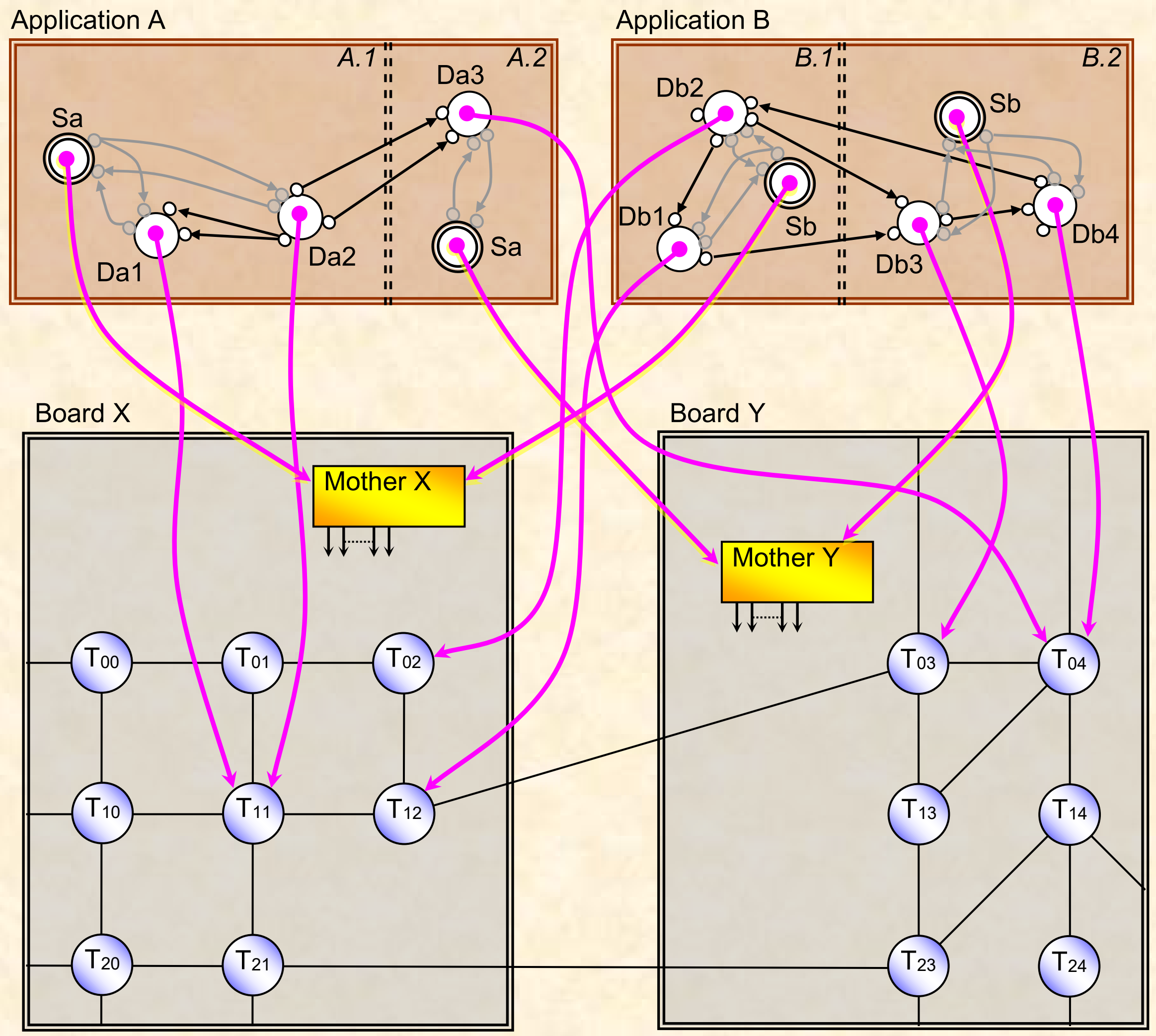


Figure III.4.3.1.a: Mapping a pair of *application* graphs to a *topology* hardware graph

- The Orchestrator contains two applications, A and B. Both these have been partitioned (by the Orchestrator) into two **application partitions**: {A.1, A.2} and {B.1, B.2}.

- A1 and B1 are mapped to hardware board X, A2 and B2 are mapped to board Y.

- ***Device mapping:*** All the devices in A1 and B1 are mapped to threads on board X; the devices in A2 and B2 are mapped to threads on Board Y. (The mapping process is described later.)

- ***Supervisor mapping:*** Associated with each application is a **supervisor**. This is a component that provides the next level in the abstract application compute hierarchy. Exactly one supervisor definition (but many instances) is permitted with each application. Permitted, not required, because a default supervisor functionality is provided by the Orchestrator that is augmented by any application defined module.

- ***Device/supervisor communication:*** Each partition of each task has a copy of the relevant supervisor. (A.1 and A.2 both have separate, but identical instances of Sa; likewise task B.) Each supervisor has just two pins - input and output, and each device in a partition has two-way communication with the **partition supervisor instance** (shown in grey in the figure).

- The Orchestrator provides a Mothership process (part of the MPI Universe) for each board. This contains a **supervisor shell** object. The supervisor shell supports the default application supervisor behaviour; any application-defined augmentation is realised as a DLL (Dynamic Link Library in Windoze-speak) or SO (Shared Object in Linux speak), which the shell accesses. In the figure, the supervisor DLL in Mothership X is created (by the Orchestrator) from the supervisor definition in application A and the supervisor definition in application B - likewise the supervisor DLL on Board Y.

A few points in general about figure III.4.1.b:

- The structure is built upon the foundation of instances of `pdigraph`. This is a multidimensional extension of the STL container {key:data} concept. The `pdigraph` contains a digraph {vertex, edge, pin}, all of which may be accessed quickly (logarithmic time) via a PoD ***key***. (The key doesn't *have* to be PoD - pdigraph is a template - but it rather makes a mockery of the idea if it isn't). In the Orchestrator, all the keys in each graph are `unsigned`, and in keeping with the STL, *keys are not stored automatically with the data*, which is why $\mathbf{C}_{\mathbf{DEVICE}}$ et al are shown on the graph edges with a copy inside the data object.

4.3.2 Topology

The POETS project in the large is intended to embrace a sweeping spectrum of research activities. From the perspective of high-level event-based algorithm development, we might expect to start with a solid and stable hardware platform, overlay a layer of intermediate (application agnostic) software, and build from there. However, the nature (and details) of that hardware platform is itself of research interest, and it is therefore a changing target. The

Orchestrator, then, has to perform the task of providing an interface layer between an unknown hardware platform and a set of unknown domain-specific high-level applications.

This challenging functionality is met by making as much of the system as run-time

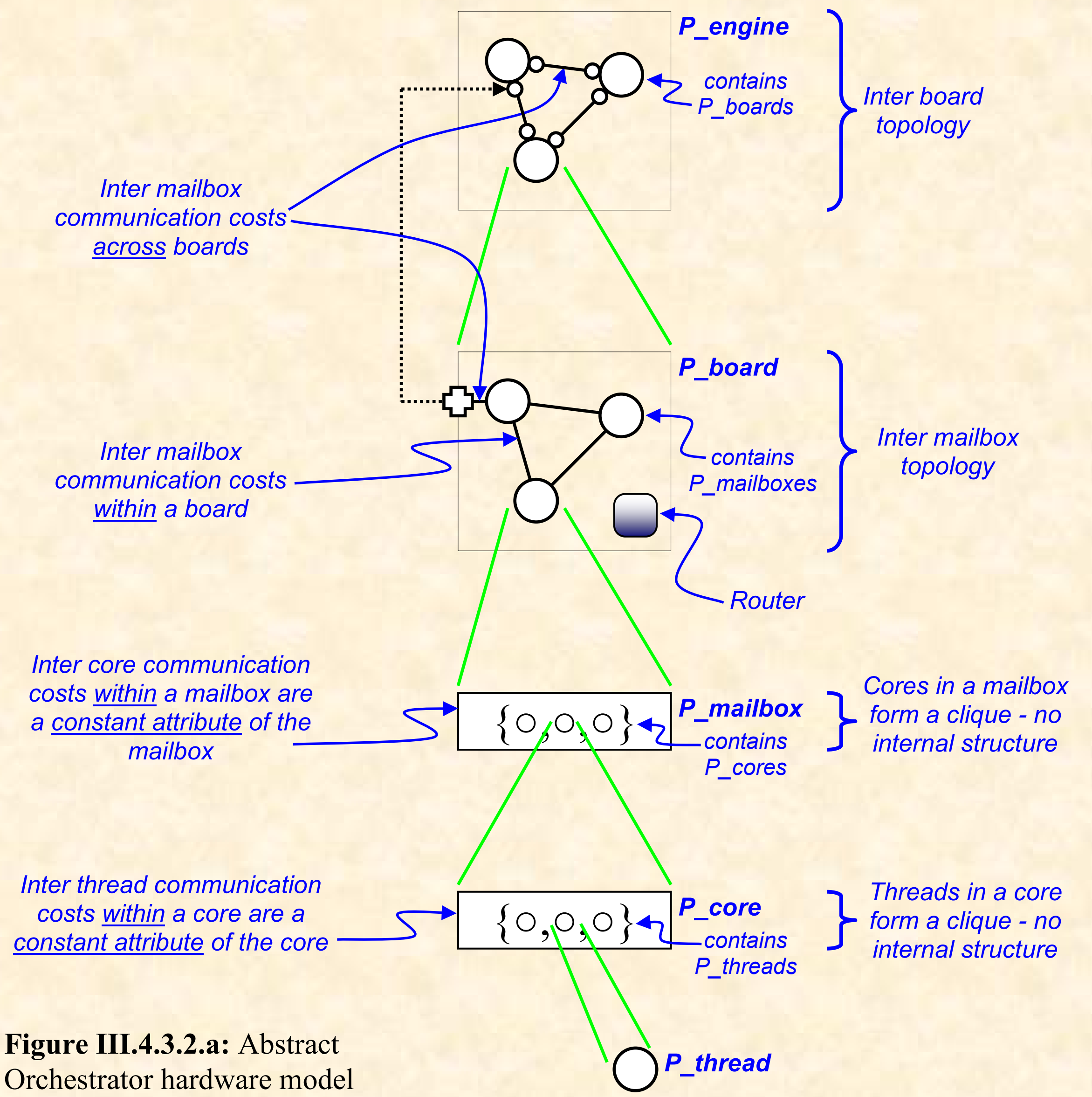


Figure III.4.3.2.a: Abstract Orchestrator hardware model

configurable as possible. Figure III.43.2.a shows an idealised representation of the Orchestrator datastructure component concerned with modelling an idealised representation of the hardware platform.

The reason for needing to capture the hardware topology to any degree of accuracy is that the outcome of any mapping subsystem (placement) will be dependent upon the relative costs of sending packet traffic over various components of the network - without an accurate and calibrated hardware model, any attempt at placement is pointless.

The area of the datastructure concerned with the topology is labelled ***platform definition*** in figure III.4.1.b. From figure III.4.3.2.a, it can be seen that a POETS engine (**`P_engine`**) contains a graph {**`P_board, P_link, P_port`**}. **`P_board`** further contain a graph of mailboxes {**`P_mailbox, P_link, P_port`**}; each mailbox contains an unordered set of cores (**`P_core`**); each core contains an unordered set of threads (**`P_thread`**). Ultimately, *multiple* devices will be mapped to *each* thread.

> Any physical system that can be represented by this structure can be configured by the Orchestrator; in corollary, any hardware build that is not capable of representation by this structure cannot be configured by (the current version of) the software.

Figure III.4.3.2.a is an idealisation; the actual Orchestrator structure (figure III.4.1.b) contains a further layer of indirection: the **box** (**`P_box`**). This plays no part in the placement calculations, but is an accurate representation of the physical implementation of the engine: engines contain boxes, which host boards, and so on. The box is in part future-proofing, and it provides a convenient and natural place to store one component of the full symbolic device address - see later. It also plays host to details of the supervisor distribution amongst the motherships.

Details of the current hardware configuration must be obtained from UoC.

4.3.2.1 Building the platform definition

The topology substructure can be built in a number of ways, namely *self (dynamic)-discovery, proof-of-life* or *a priori definition*.

Dynamic discovery

The Orchestrator datastructure can represent arbitrary connection configuration at the board and mailbox level. From the start, the design intent of the POETS system is that the Orchestrator be capable of discovering the structure of the underlying hardware autonomously; this requires interactions between the hardware and software that have yet to be implemented. This section is a placeholder for that capability.

Proof-of-life definition

The proof-of-life generator subsystem (see figure III.4.1.a and section III.4.7.4) is a mechanism whereby both the platform definition and the application trees can be populated with machine-generated (pre-baked) structures. It is a development/testing capability.

A priori definition

The platform definition substructure may be initialised from a hardware configuration file - see section III.4.6.4.

4.3.3 Applications

This section of the datastructure (device graphs and type trees) holds the abstract application graphs, created from the "compilation" of the domain-specific front end by the XML parser, the XML validator and the XML-to-datastructure builder, **DS_XML**. The XML parser and validator are generics, and have stand-alone documentation. The builder is described in section 4.3.3.3.

For historical reasons, the XML element names are not the same as the corresponding C++ class names. One might offer the justification that having different names allows different formalisms for application description to be added to the Orchestrator at a later date....

The XML correspondence is shown in figure III.4.1.b and given below:

XML element name	C++ class
Graphs	Apps_t
GraphType	GraphT_t
Properties	CFrag
Metadata	Meta_t
SharedCode	CFrag
MessageTypes	-
MessageType	MsgT_t
DeviceTypes	-
DeviceType	DevT_t
State	CFrag
SupervisorOutPin	PinT_t
OnSend	CFrag
SupervisorInPin	PinT_t
OnReceive	CFrag
InputPin	PinT_t
OutputPin	PinT_t
ReadyToSend	CFrag
OnInit	CFrag
OnHardwareIdle	CFrag
OnDeviceIdle	CFrag
ExternalType	DevT_t
SupervisorType	SupT_t:DevT_t
Code	CFrag
OnSupervisorIdle	CFrag
OnRTCL	CFrag
OnStop	CFrag
OnCTL	CFrag
GraphInstance	GraphI_t
DeviceInstances	-
DevI	DevI_t
ExtI	DevI_t
EdgeInstances	-
EdgeI	EdgeI_t

4.3.3.1 Devices, supervisors and externals

Currently, three types of computing element exist in a POETS application: Normal devices (**D**), Supervisors (**S**) and Externals (**E**).

- D (normal) devices are the compute elements that are realised in the hardware platform.

- E (external) devices are realised external to the system and on deployment, interact with the NameServer (section III.5).

- S (supervisors) are outlined in section III.4.3.1 and are described in detail in section III.12.

From the perspective of the datastructure, both D and E devices are represented by the **`DevI_t`** class, the only difference being the values of the **`char devTyp`** ('D'/'X'/'S'/'U').

Supervisor devices are application specific, and hence must be defined by the application XML. (The Orchestrator provides a *default* supervisor for each application. The functionality of this is simply and solely to pass information blindly through the mothership to and from the packet network.) Each application must have exactly one supervisor defined (which may be the default). The description is held in **`SupT_t:DevT_t`**, and anchored as a single virtual supervisor device held in **`GraphT_t::pSup`**. (See figure III.4.1.b.)

A non-intuitive aspect of the structure is outlined in figure III.4.3.3.1.a. The XML input graphs allows multiple path channels attached to a single pin; the **`pdigraph`** template allows only one. This is overcome by use of indirection in the datastructure, as shown in the figure.

4.3.3.2 Type trees

The application graph is strongly typed: a graph has a type (**`GraphT_t`**), graphs contain devices which have types (**`DevT_t`**), devices contain pins which have types (**`PinT_t`**), and pins reference messages which have types (**`MsgT_t`**). The structure - a tree - is generated

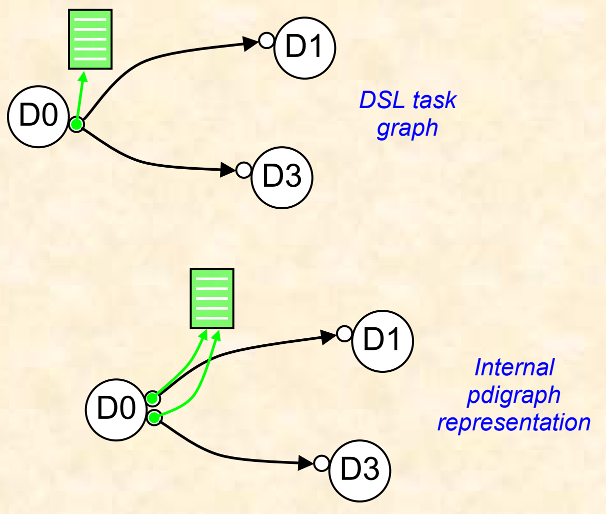


Figure III.4.13: Internal representation of a multi-path channel in a Domain-Specific-Language abstract application graph.

naturally from the hierarchical nature of an XML description, although the POETS XML definition has more levels in the hierarchy than are necessary or realised here.

Each type definition has a string ***name***, that allows component ***instantiations*** to be linked to their associated type definition.

All of the objects in the type trees are derived from **`DefRef`** and **`NameBase`**, but not **`DumpChan`** because none of them are stored in a pdigraph template.

Graph types and device types can contain metadata, and all of them have numerous pointers to **`CFrag`** objects (♦[...]) in figure III.4.1.b). These (CFrags) are holders for the fragments of C code (state, properties and event handlers) that are embedded in the generating XML to be inserted into the softSwitch and supervisor skeletons and deployed to the hardware platform.

Whilst the XML is checked for both syntax and semantics (section 4.5) the embedded C is not parsed in any way: it is treated simply as character strings and no checking is performed on it until the assembled code is handed out to the third-party cross compiler.

The cross-compiler diagnostics will be in terms of the C-source line number presented to it, which will have a lot of Orchestrator-generated chaperone code interspersed. The Orchestrator therefore *decorates* the C fragments handed to the cross-compiler. If the C fragment is not empty, the comment

`// Line` *line*

is prepended to the code. If a CDATA XML element is expected but not supplied, the validator will complain; however, if the element is supplied but empty (which may be valid behaviour), a further C comment is added:

`// Line` *line*
`// CFrag` *full hierarchical XML element name* empty

(The element name may be quite lengthy:

`../TestData/adb/mlv_03_f.xml.Graphs.GraphType.DeviceTypes.DeviceType.InputPin.OnReceive`)

(The separator '.' between the file name **`mlv_03_f.xml`** and the XML hierarchy is misleading, but it's generated automatically from the **`NameBase::FullName()`** method, and burgling the process just to change a delimiter once didn't seem worthwhile.)

Recall that the line numbers generated by the XML subsystem always refer to the line number on which the relevant element *ended*, **not** when it started.

In the event of compiler errors, these comments will enable the user to trace the errors back from the cross-compiler to the user source code.

4.3.3.3 The instantiation graph

As the name implies, the instantiation graph section of the application definition is rooted in a pdigraph template, populated by the datastructure builder outlined in the next section.

`Pdigraph` is a template designed along the same lines as an STL container. It holds a tripartite graph: {device, edge, pin}, corresponding to the classes **`DevI_t`**, **`EdgeI_t`**, **`PinI_t`** in the table in the previous section.

Because there is significant element copying, the graph elements are dynamic, and only the addresses are actually inserted into the container:

```
 pdigraph<unsigned,DevI_t *,unsigned,EdgeI_t *,unsigned,PinI_t *> G;
```

Each of the three elements is addressable by an **`unsigned`** key. **`Pdigraph`** access functions (**`NodeKey(key)`**, **`ArcKey(key)`** and so on) give access to the data *stored in the digraph*. This data is the object *address*, so to get to the actual data, the value returned by the access functions has to be further dereferenced: **`*NodeKey(key)`**.

The *value* of the key is also stored in the element concerned (**`unsigned EdgeI_t::Key`**, **`unsigned DevI_t::Key`** and **`vector<unsigned>PinI_t::Key_v`**). The Pin instance key is multi-valued, explained in section 4.3.3.4; multiple graph keys can return an index to the same pin instance object. The datastructure underlying this is shown in the inset in figure III.4.1.b and III.4.3.3.3.a.

The graph contains node elements for both external devices (**`DevI_t::devTyp = 'X'`**) and 'internal' devices (**`DevI_t::devTyp = 'D'`**); these are interconnected by edges in the normal way. Supervisor device ('S') instances are not stored in the graph - their deployment is part of the function of the placement subsystem, and not defined by the XML specification. Equally, every device is assumed to be in two-way communication with a supervisor, but this

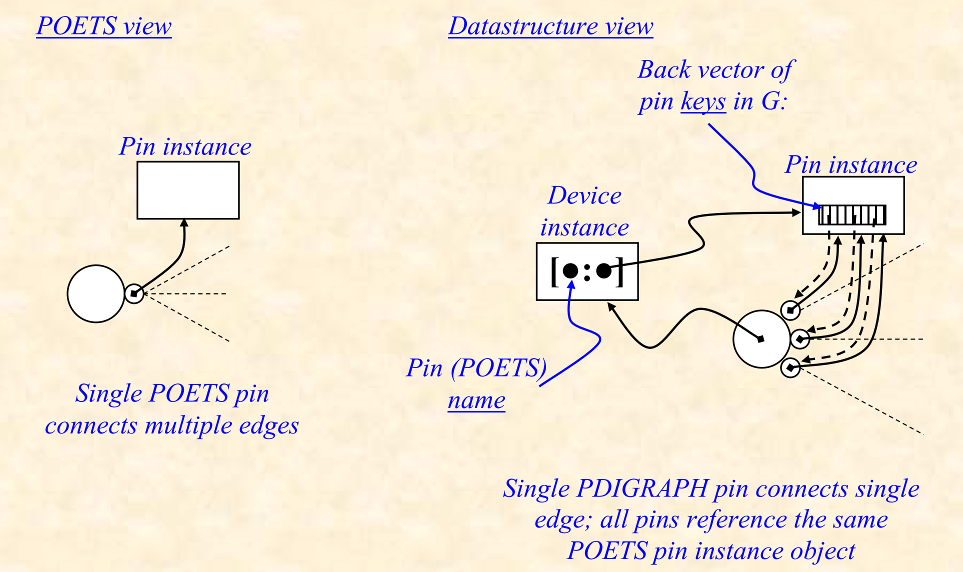


Figure III.4.3.3.3.a: Scaffold map for pin structure assembly

connectivity is not stored in the graph either, because it is ubiquitous.

`DevI_t::devTyp` is constructed as '`U`' (uninitialised), and set to 'X'|'D' during the datastructure build. Any value !='`X`' or '`D`' is an internal error. Note that the corresponding entity type definitions (`DevT_t` and `SupT_t`) *also* have a `devTyp` field. Here, `devTyp` may also take the value '`S`' - supervisor.

The graph instance object contains a `map<string,unsigned> Dmap` that links the device names as known by the XML to the device graph keys (and thence to the device instance itself). This is here to facilitate `NameServer` interaction and data exfiltration.

The device instance class `DevI_t` contains a `map<string,PinI_t *>Pmap`. This is a map linking the device-local pin names to the pin instance objects. It is a scaffolding container, used in the generation of the pin key vector in `PinI_t`. Containing strings, it is potentially quite a large component of the build memory footprint. *For this reason it is cleared at the end of the graph build stage.*

4.3.3.4. DS_XML - the datastructure builder

The datastructure loader/builder is invoked from the command line (*load /app=filename*) via the command handler `CmLoad::Cm_App`. After checking for existence, collisions and so on, the file is parsed and validated (`XValid::Validate(`*filename*`)`) and the datastructure build proper is invoked as `DS_XML::PBuild(xnode *` *pn*`)`, where *pn* is the root of the (successfully validated) XML node tree. The XML tree is a domain-agnostic token tree of symbols reflecting the information contained within the original input file.

It is at this point that the generic XML tree is used to create the strongly-typed Orchestrator database. The structure defined in the grammar configuration file is reconciled with the code in `DS_XML` - ***it follows that if the one is changed, the other must be altered also.*** `DS_XML` contains rudimentary defences: if an unexpected XML subelement is found, an Unrecoverable error will be posted (because the unexpected element should have been caught by the Validator). `DS_XML` contains few checks for other semantic errors: duplicate entity definitions are the exception. These are flagged (as warnings) and definitions subsequent to the first ignored. `DS_XML::PBuild` oversees a depth-first walk of the type agnostic node tree. The element name indicates what set of subelements may occur (this has been checked by the Validator, so there will be nothing there that shouldn't be there. However...), and this set of subelements drives a node-local switch statement. Effectively, we need a switch statement with a string control variable. (Provided as native in PL/1, but not, alas, C++). However, the set of allowable subelement names at each node is small and known a priori, and the DS_XML constructor builds a `map<string,enumerated_type> DS_map` to turn the element name into an integer type. This, in turn, drives a switch statement in every element-specific function in `DS_XML`.

Transforming the XML type definition tree into the Orchestrator type definition tree is simple; the latter is constructed on the fly as the former is traversed. Figure III.4.3.3.4.a depicts a datastructure snapshot as the type definition part of the XML tree is traversed (the simplest component). The call stack is

```
[ 0] main(int,char *)
[ 1] Root::Root(int,char *,string)
[ 2] CommonBase::MPISpinner()
[ 3] Root::OnIdle()
[ 4] Root::ProcCmnd(Cli *)
[ 5] CmLoad::operator()(Cli *)
[ 6] CmLoad::Cm_App(Cli Cl_t)
[ 7] DS_XML::PBuild(xnode *)
[ 8] DS_XML::_Apps_t(xnode *)
[ 9] DS_XML::_GraphT_t(Apps_t *,xnode *)
[10] DS_XML::_MsgT_ts(GraphT_t *,xnode *)
[11] DS_XML::_MsgT_t(GraphT_t *,xnode *)
[12] DS_XML::_CFrag(xnode *)
```

`xnode` is typedefed from **`xmlTreeDump::node`**.

Notice stack frame [3] **`Root::OnIdle`**: the test data generating this example is run from a batch file.

`DS_XML::PBuild` starts the client XML tree traversal proper. Broadly, each of the **`_***`** functions is handed a tree node address, and it creates and attaches an Orchestrator structure accordingly, then walks the XML tree nodes to handle child elements. The Orchestrator datastructure is a tree; **`PBuild`** et seq simply walks (depth first) the homogeneous POETS-agnostic XML node tree, creating the strongly typed heterogeneous Orchestrator tree as is goes.

Creating the instance graph (figure III.4.3.3.4.b) is a little more nuanced: Neither pins nor edges are explicitly declared as stand-alone entities, and edge instances (**`EdgeI`**) and device instances (**`DevI`**, **`ExtI`**) may occur in any order. If a reference to a device occurs *before* it is defined, that reference is interpreted as a forward declare, and the device inserted into the graph, but with **`DevI_t:DefRef`** set to indicate the outstanding definition. When the definition is encountered, **`DefRef`** is updated. Any unsatisfied **`DefRef`** tags outstanding at the end of the **`GraphInstance`** element will be flagged to the user by **`GraphI_t::UndefDevs()`** called from **`DS_XML::_GraphI_t()`**.

The call stack for figure III.4.3.3.4.b is

```
[ 0] main(int,char *)
[ 1] Root::Root(int,char *,string)
[ 2] CommonBase::MPISpinner()
[ 3] Root::OnIdle()
[ 4] Root::ProcCmnd(Cli *)
[ 5] CmLoad::operator()(Cli *)
[ 6] CmLoad::Cm_App(Cli Cl_t)
[ 7] DS_XML::PBuild(xnode *)
[ 8] DS_XML::_Apps_t(xnode *)
[ 9] DS_XML::_GraphI_t(Apps_t *,xnode *)
[10] DS_XML::_DevI_ts(GraphI_t *,xnode *)
[11] DS_XML::_DevI_t(GraphI_t *,xnode *)
```

Pins have device-local names, and an edge is defined as a *device:pin-device:pin* tuple (see next section). For historical reasons, edge definitions where one or both device fields are blank are intended to represent supervisor connections. However, subsequently it has been decided that *every device has an implied supervisor connection, so there is no need to represent it explicitly in the application definition*. Thus is follows that currently, supervisor connections are ignored by **`DS_XML`**.

You may be curious about the large slabs of whitespace appearing in this document. You'd have to ask the software folk at Micro$oft Office HQ about that. It's all a mystery to me.

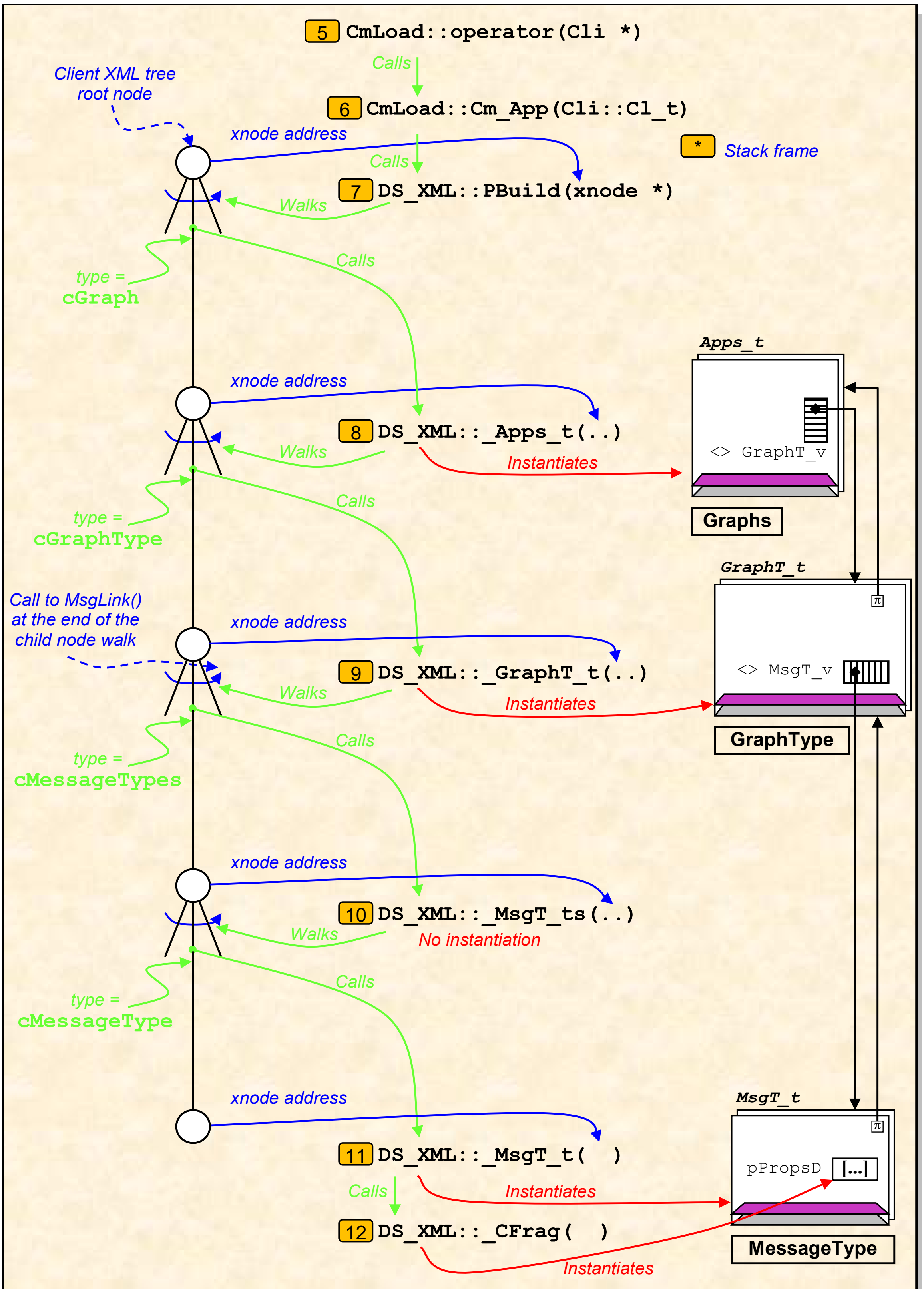


Figure III.4.3.3.4.a: Call path of the datastructure build - type tree branch

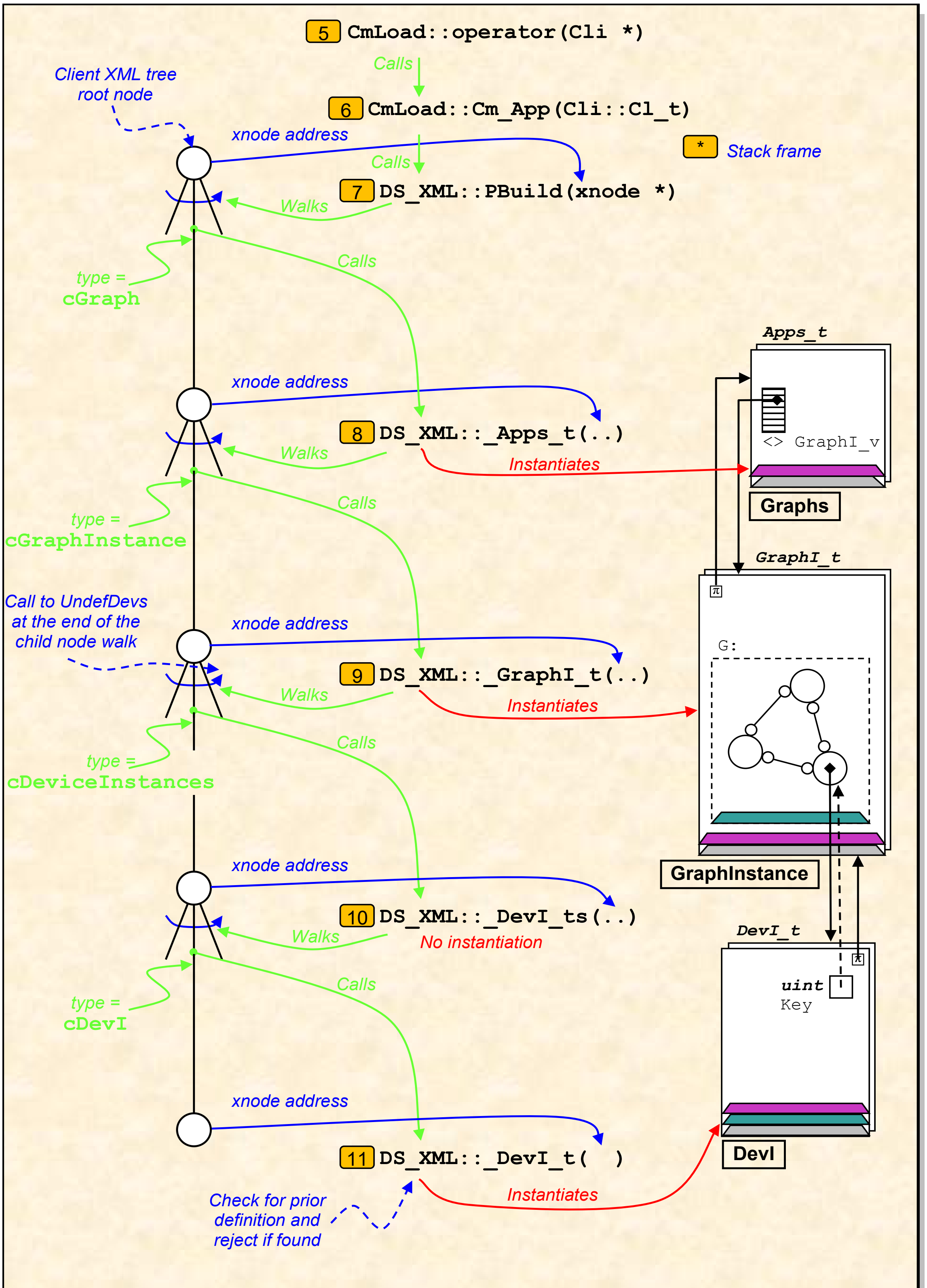


Figure III.4.3.3.4.b: Call path of the datastructure build - graph instantiation

4.3.3.5 **`pathDecode`**

The POETS **`EdgeI`** element has an attribute called *paths*, whose value (from the perspective of XML) is a string. This string has an internal structure, which (from the perspective of POETS) needs to be interpreted, and checked for syntactic integrity.

The string is (should be) of the form

```
    to_device : to_pin - from_device : from_pin
```

Either (but not both) device fields may be empty, implying supervisor connections.

This is parsed by a functor **`pathDecode()`**, that returns a string vector. The vector has at least one element:

> [0] empty if the functor argument parsed OK, otherwise the *STRING* "1", "2" and so on, indicating the *position* of the error in the string. In the case of a syntax error, this is the sole element of the vector.

If the string parsed to a path without error, the vector contains five elements:

> [0] empty
> [1] **`to_device`** (may legitimately be empty)
> [2] **`to_pin`** (never empty)
> [3] **`from_device`** (may legitimately be empty)
> [4] **`from_pin`** (never empty)

The syntax transition graph is shown in figure III.4.3.3.5.a.

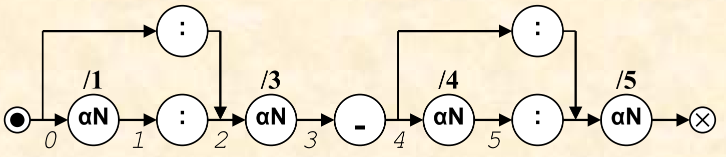


Figure III.4.3.3.5.a: Inner syntax transition graph for **`pathDecode`**

Token	Symbol
t0	α_N
t1	:
t2	-
t3	else

4.3.4 Crosslinks

Thus far, the skeleton of figure III.4.1.b has been described: hardware description tree, application instance graphs, type definition trees. These are the links that are traversed during tear-down.

Three further sets of linkage exist within the datastructure; two mutable by the user, one not.

4.3.4.1 M(essage) links

All the pins in a type tree (device and supervisor input and output) have a message type associated with them. This is little more than a `CFrag`, anchored in a `MessageType` (see figure III.4.3.4.1.a). At the end of the `GraphType` build (`DS_XML::_GraphT_t()`), the

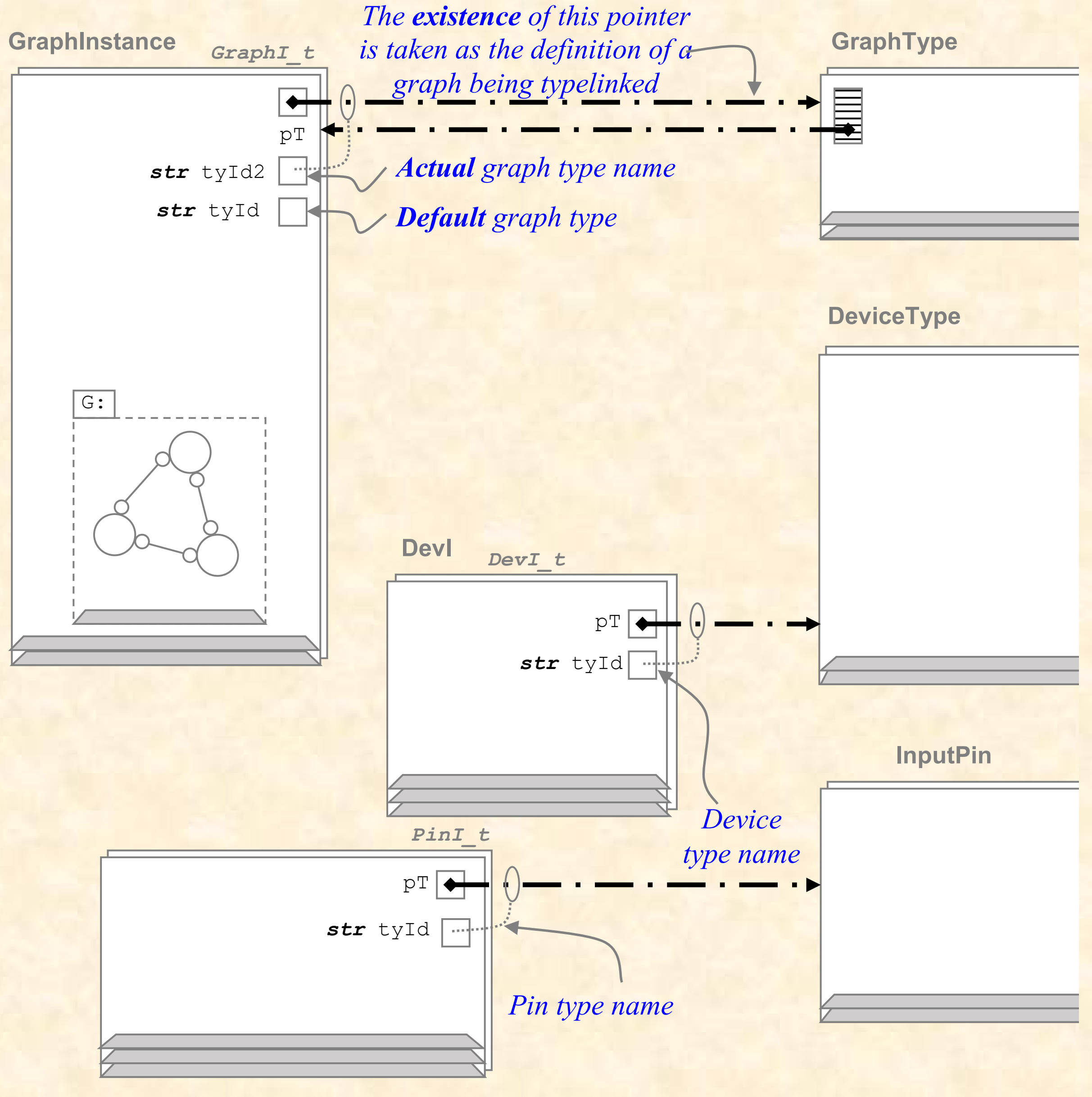


Figure III.4.3.4.2.a: Typelinking an application

"messagelink" pointers are created by a call to **GraphT_t::MsgLink()**. This links the name of a message in **PinT_t::tyId** to the actual instance of that message in **MsgT_t**. Note that this can never fail: if a pin cannot find a named message, a default message is returned (named "**]default[**" - hard to override accidentally). This is pre-emptively created and loaded into the first [0] element of **GraphT_t::MsgT_v** by **DS_XML::_GraphT_t()** - the footprint is extremely small. This *default* CFrag contains

```
// Auto-generated default POETS packet format
uint8_t [56] payload;
```

These links are inserted automatically when the application is loaded; they are not user-mutable.

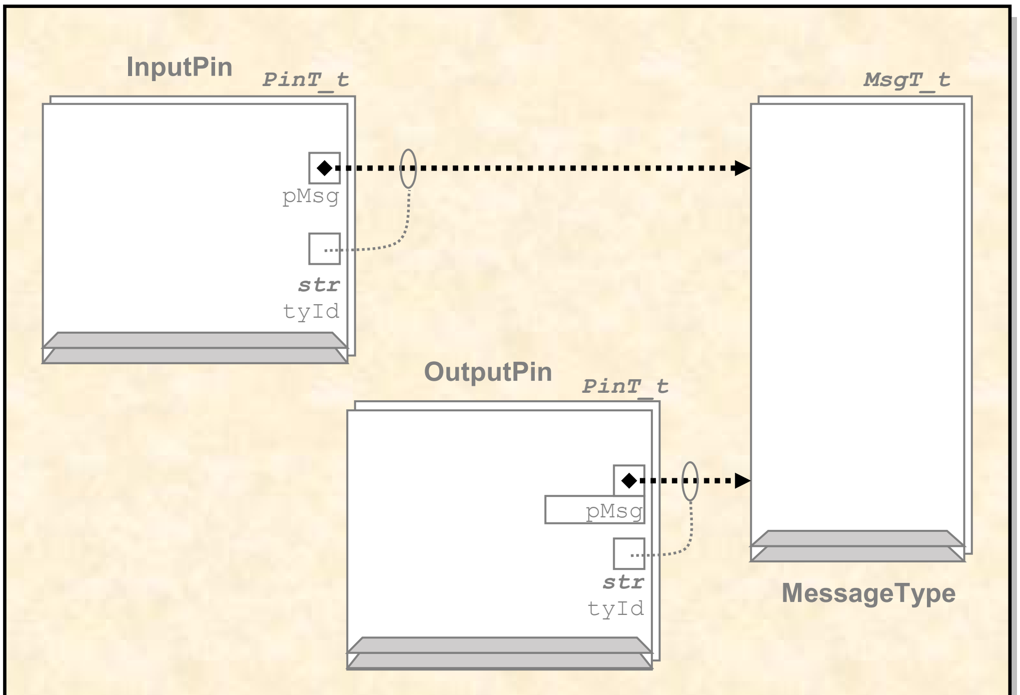


Figure III.4.3.4.1.a: Messagelinking an application

4.3.4.2 T(ype) links

The **typelinks** refer to the cross links connecting the graph-, device- and pin instances to their respective types - see figure III.4.3.4.2.a. They are not created automatically on application load, they must be inserted manually using the **`tlink`** command. By default, a graph instance will link to the graph type tree referenced within it, but this can be changed using **`tlink`** (see section 4.10.15). These links must be in place before the binary build takes place. It is useful, but not mandatory, for them to be in place before placement (so that the placement subsystem can get an idea of the memory footprint required for the various device types). Further details are supplied in the command description section.

4.3.4.3 P(lacement) links

The placement links indicate the mapping between abstract device instances and the hardware that will support them - see figure III.4.3.4.3.a. The placement subsystem is described in detail in separate documentation: "**placement**".

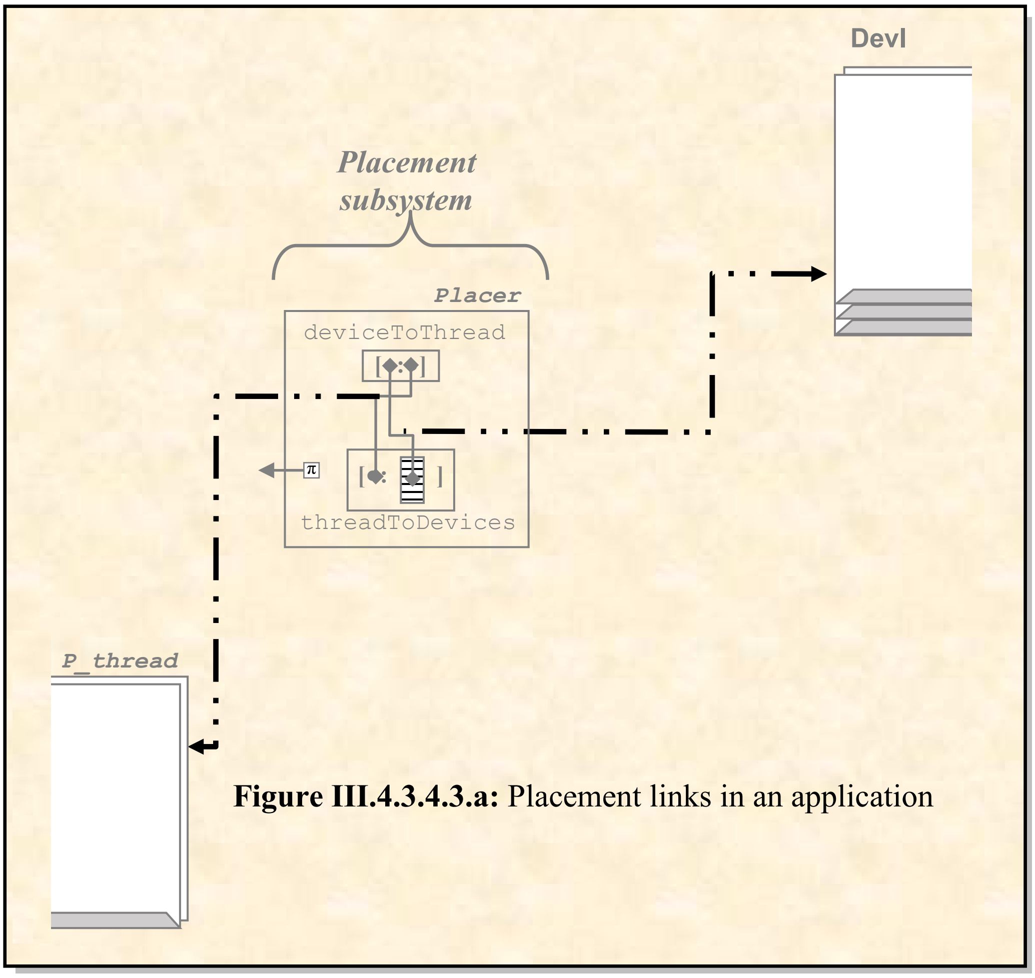


Figure III.4.3.4.3.a: Placement links in an application

4.4 Placement subsystem

4.5 XML processing

4.5.1 Supported subset

4.5.2 Parse tree

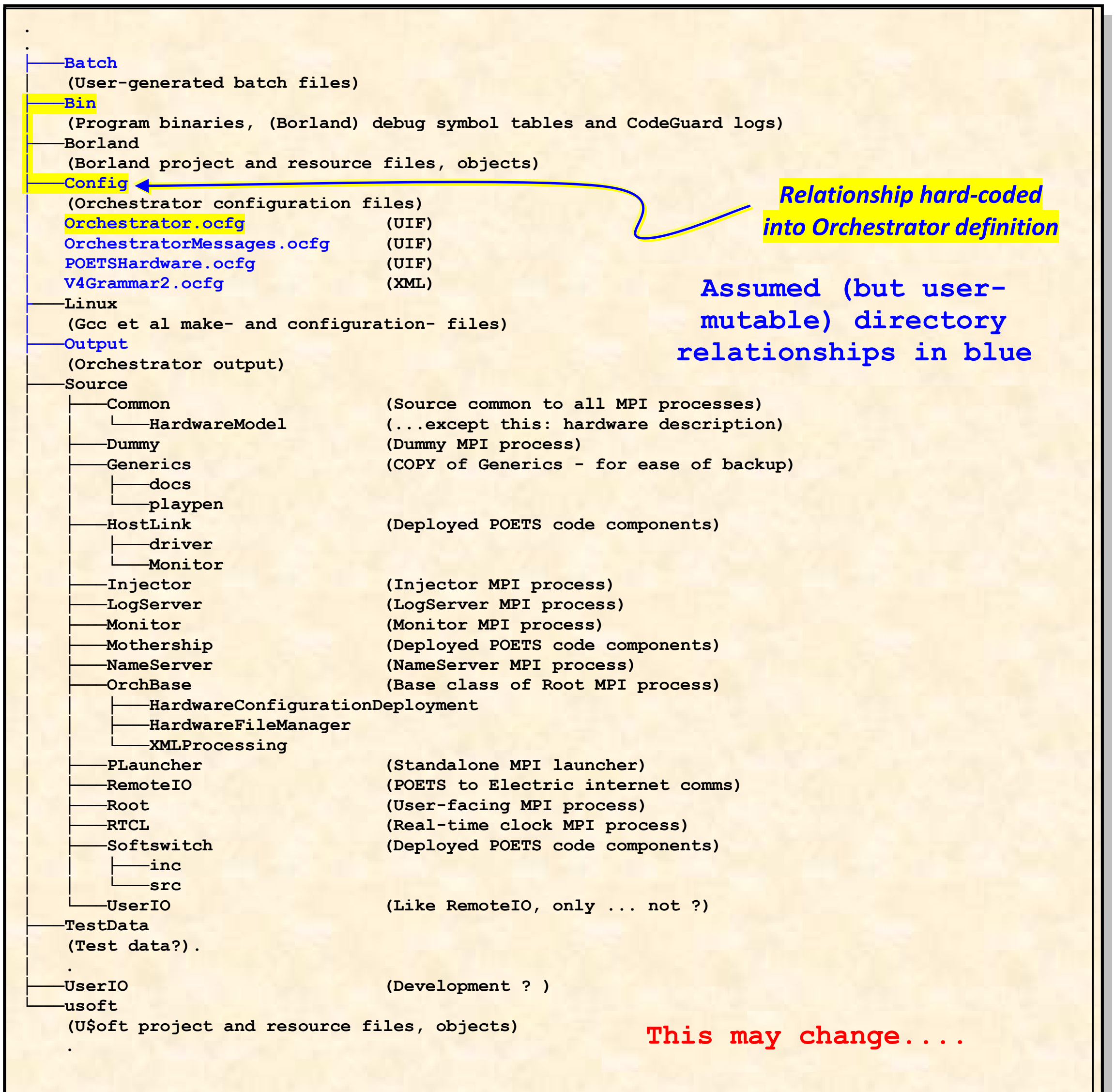
4.5.3 Validation

Documented separately (Validator.doc)
Moving this (XValid) to generics - possibly. When I find the time.
Recast pathDecode from a functor, and make it a flat C function. It has no internal state, and the transition matrix can be made static if anyone wants. Then we can put XValid into generics, and attach pathDecode as a callback function at runtime. Job done. Tomorrow.

4.6 Configuration files and layout

4.6.1 Directory and file layout

The file and directory layout is something like that shown in figure III.4.6.1.a.



```
.
.
├──Batch
│   (User-generated batch files)
├──Bin
│   (Program binaries, (Borland) debug symbol tables and CodeGuard logs)
├──Borland
│   (Borland project and resource files, objects)
├──Config
│   (Orchestrator configuration files)
│   Orchestrator.ocfg              (UIF)
│   OrchestratorMessages.ocfg      (UIF)
│   POETSHardware.ocfg             (UIF)
│   V4Grammar2.ocfg                (XML)
├──Linux
│   (Gcc et al make- and configuration- files)
├──Output
│   (Orchestrator output)
├──Source
│   ├──Common                      (Source common to all MPI processes)
│   │   └──HardwareModel           (...except this: hardware description)
│   ├──Dummy                       (Dummy MPI process)
│   ├──Generics                    (COPY of Generics - for ease of backup)
│   │   ├──docs
│   │   └──playpen
│   ├──HostLink                    (Deployed POETS code components)
│   │   ├──driver
│   │   └──Monitor
│   ├──Injector                    (Injector MPI process)
│   ├──LogServer                   (LogServer MPI process)
│   ├──Monitor                     (Monitor MPI process)
│   ├──Mothership                  (Deployed POETS code components)
│   ├──NameServer                  (NameServer MPI process)
│   ├──OrchBase                    (Base class of Root MPI process)
│   │   ├──HardwareConfigurationDeployment
│   │   ├──HardwareFileManager
│   │   └──XMLProcessing
│   ├──PLauncher                   (Standalone MPI launcher)
│   ├──RemoteIO                    (POETS to Electric internet comms)
│   ├──Root                        (User-facing MPI process)
│   ├──RTCL                        (Real-time clock MPI process)
│   ├──Softswitch                  (Deployed POETS code components)
│   │   ├──inc
│   │   └──src
│   └──UserIO                      (Like RemoteIO, only ... not ?)
├──TestData
│   (Test data?).
│
│   .
├──UserIO                          (Development ? )
└──usoft
    (U$oft project and resource files, objects)
    .
```

Relationship hard-coded into Orchestrator definition

Assumed (but user-mutable) directory relationships in blue

This may change....

Figure III.4.6.1.a: Orchestrator directory and file layouts

- Shown is the entire development directory tree. The blue subtree is probably the only necessary deployment structure.

- The Orchestrator binaries *require* that the file **`Orchestrator.ocfg`** resides in the directory **`Config`** with a set relationship to the binaries. This is defined near the top of **`..\Source\Root\OrchConfig.cpp`**:

```
const string OrchConfig::WhereAmI = string("../Config/Orchestrator.ocfg");
```

- The locations of the other configuration files (`OrchestratorMessages.ocfg`, `POETSHardware.ocfg` and `V4Grammar2.xml`) are defined *within* the file `Orchestrator.ocfg` and so can be changed without recompiling.

4.6.2 Orchestrator configuration `Orchestrator.ocfg`

The Orchestrator configuration file sits at the top of the configuration file hierarchy. A typical example is shown in figure III.4.6.2.a.

```
  // ORCHESTRATOR CONFIGURATION FILE
  // NOT user-facing
  // This is the root of the configuration file hierarchy. It contains the locations
  // of all necessary setup files *relative* to the root binary.
  // Apart from itself, obviously: the location of *this* file is hardwired
  // into the code (OrchConfig.cpp after the header #includes) as
  // static const string OrchConfig::WhereAmI = string("../Config/Orchestrator.ocfg");

[Orchestrator_header]
  // All pretty arbitrary; just copied in and stored
name        = OrchestratorConfiguration
author      = TheSupremeBeing
date        = "5/11/19"
version     = "0.0.1"

[default_paths]
  // All these may be overridden by the console "path" command
  // Application (XML) files
apps
  // Hardware description files. Only used on reload; not visible to the launcher
engine      =
  // Placement control; default algorithm + control parameters
place       = "Wally the dog"
  // Logserver output (sent to LogServer on startup)
log         = "../Output/POETS_Logfile.log"
  // Microlog output
//ulog        = "E:\Grants\POETS\Software\ZeOrchestratings\Borland\Ulog.log"
ulog        = "E:\Grants\POETS\Software\ZeOrchestratings\Output\"
  // Trace output
trace       = "     "
  // Softswitch binary files for execution
binaries    = "      "
  // Orchestrator generated source and X-compiled output prior to deployment
stage       = "    "
  // Batch files
batch       =
  // Deployed supervisor binaries (in the MPI universe)
supervisors = "    "

[setup_files]
  // Note these are resolvable files, not paths
  // Elaboration messages for the user via the LogServer
messages    = "../Config/OrchestratorMessages.ocfg"
  // XMLapplication grammar validation definition
grammar     = "../Config/V4Grammar2.ocfg"
  // Hardware definition file
hardware       = "../Config/POETSHardware.ocfg"
  // Startup algorithm + control parameters
placement   = "the frumious bandersnatch"

[flags]
  // Default flags for the cross-compiler
build       = "\oink -plop ++wheeee  !. <?> "

[errors]
  // Elaboration messages passed out to the author for errors in processing THIS file
  // They are all classed as "Unrecoverable", not because they are, but because:
  // If you're not a grown-up you shouldn't be mucking about with it anyway
  // If you are a grown-up you'll know how to fix it without further help
  // And then.... I realised that if there's a syntax error in *this* file the records
  // below won't get stored anyway, so it's all a bit pointless.
  // There is, therefore, no code behind this section. Just treat it as a reference map
  // of error numbers (that *are* reported) : meanings.
001(U)  : "Section has != 1 names"
002(U)  : "Variable in section [Orchestrator_header] has too many values"
003(U)  : "Variable in section [default_paths] has too many values"
004(U)  : "Variable in section [setup_files] has too many values"
005(U)  : "Variable in section [flags] has too many values"
006(U)  : "Configuration file inaccessible"
007(U)  : "Configuration file corrupt (syntax)"

//-----------------------------------------------------------------------------
```

Figure III.4.6.2.a: Orchestrator configuration file

4.6.2.1 Orchestrator configuration section : [**Orchestrator_header**]

Intended to support some sort of provenance control; jam tomorrow.
There is no code under this section.

4.6.2.2 Orchestrator configuration section: [**default_paths**]

This section defines the *default* values for the various filepaths used by the user. They may be absolute or relative. Linux uses '/' for directory path delimiters, u$oft by default uses '\', although it is tolerant of both, so irrespective of what you type, the Orchestrator turns everything into Linux-compliant path names delimited with '/'.

A path may be reset to its default value by supplying '~' instead of a path string. For example, modifying the path for the micrologs:

```
path /ulog = "\stuff/whiffle\"  // Set path relative to the Orchestrator binary
path /ulog                      // Set to the console
path /ulog = ~                  // Set path to value supplied in Orchestrtor.ocfg
```

There are many console commands in which the Orchestrator expects a filename to be supplied. If the filename is supplied undecorated, in Windows, subdirectories will be searched according to the PATH global variable. If the filename is prepended with a '+', the relevant file path (it is context sensitive) will be prepended to the name. User-facing details are provided in section 4.10.

4.6.2.3 Orchestrator configuration section : [**setup_files**]

This section defines the locations of the rest of the setup files for the Orchestrator, consistent with section 4.6.1. As long as the names and filepaths in this section are consistent, the Orchestrator is indifferent to the actual locations. The **`.ocfg`** extension is for convention only.

4.6.2.4 Orchestrator configuration section : [**flags**]

This section defines the control flags for the cross-compiler.

4.6.2.5 Orchestrator configuration section : [**errors**]

See the comments embedded in the configuration file, explaining why this section is pointless. Oops.

4.6.3 Orchestrator Logserver messages `OrchestratorMessages.ocfg`

This is the file that contains the full enumeration of every possible message posted by the Orchestrator. The functionality is described in section III.7. On startup, the Root process will send a message containing the location of the file to the LogServer, which will read it once and close it.

4.6.4 POETS hardware description `POETSHardware.ocfg`

An example is given in figure III.4.6.4.a; full details are documented separately: see "**hardware_description_file_and_reader**" and "**hardware_model**".

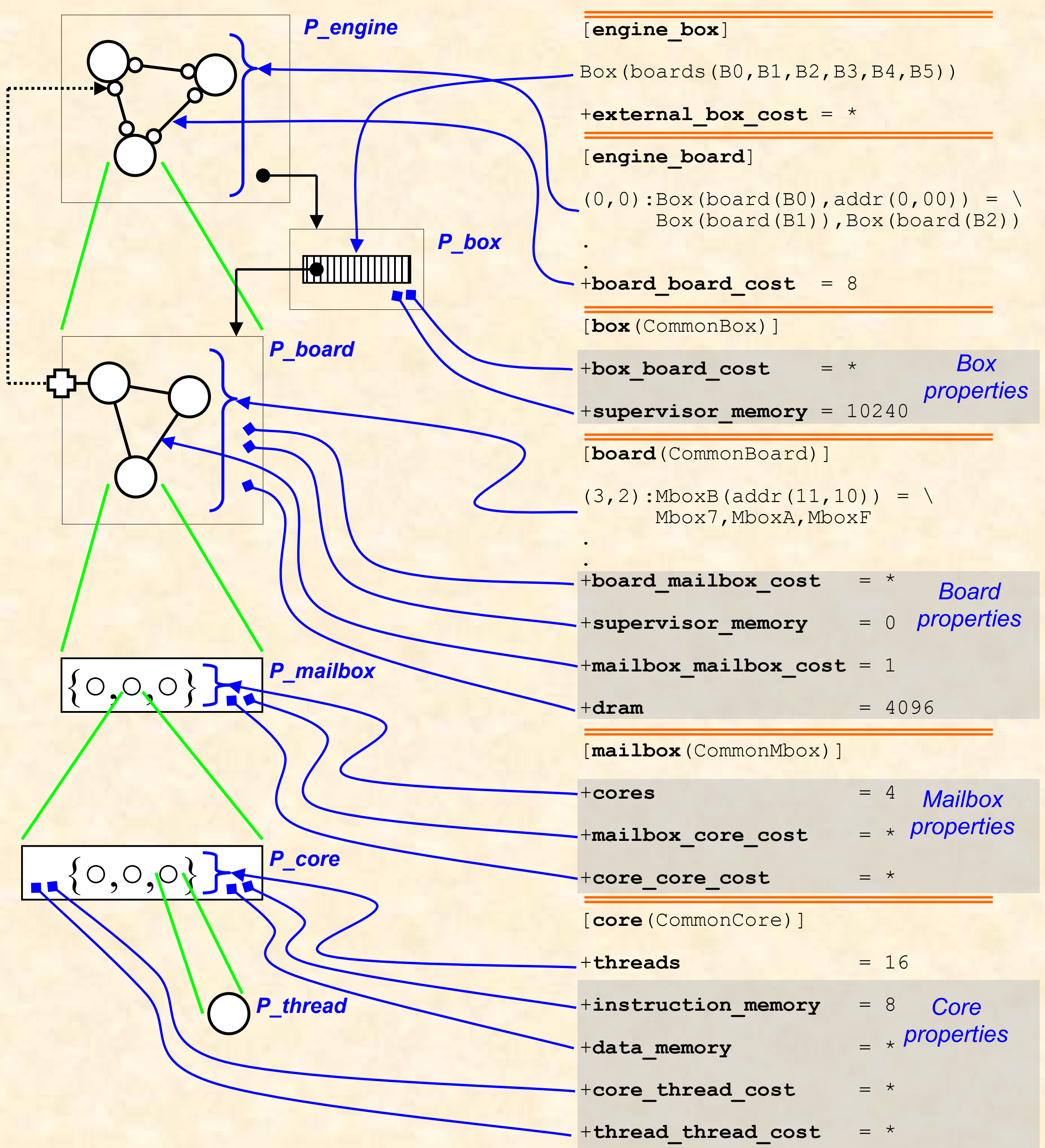
The file supports description of all the salient metrics relevant to placement (+**`xxxx_xxxx_cost`** *et al*); the topologies of the boards within engines (section **`engine_board`**) and mailboxes within boards (section **`board`**).

The file format supports a number of different ways of defining topologies; the dialect of figure III.4.6.4.a allows the explicit enumeration of every graph connection, but it is alternatively possible to simply instantiate uniform meshes of arbitrary dimensionality and other structures.

Figure 4.6.4.b shows the relationship between the file contents and the OrchBase data structure.

```
[header(Coleridge)]                       // Provenance control section
+author           = "Mark Vousden"        // Guess
+dialect          = 3                     // Internal consistency
+datetime         = 20181127165846        // yyyy mm dd hh mm ss
+version          = "0.3.0"               // Format version
+file             = "dialect_3.ptop"      // Name of this file

[packet_address_format]                   // Define internal hardware address component
+mailbox          = (2,2)                 // Components can have inner structure
+thread           = 4                     // 4 bits = 0..15 thread ids
+core             = 2                     // = 0..3 core ids
+board            = (2,2)                 // = (0..3,0..3)
+box              = 0                     // There is only one box in Coleridge so this
                                          // address component is valid (=0)

[default_types]                           // If not explicitly specified in topologies
+box_type         = "CommonBox"           // Components all have named types
+board_type       = "CommonBoard"         // (except threads)
+mailbox_type     = "CommonMbox"
+core_type        = "CommonCore"

[engine_box]                              // Define engine-box relationship
                                          // An enumeration of the boxes in engine:
Box(boards(B0,B1,B2,B3,B4,B5),hostname(machine_1))
Crate(boards(Ba,BB,BX),hostname(machine_2))
+external_box_cost = *                    // <!> Missing, used for externals.
+box_box_cost      = 1                    // Unused (there's only one box here!)

[engine_board]                            // Define engine-board relationship
// Layout:                                // This is topology of boards in engine
//   0 -- 1
//   |    |
//   2 -- 3                               // Board *type* not specified here because
//   |    |                               // the instance picks up the default type
//   4 -- 5                               // (here "CommonBox")
(0,0):Box(board(B0),addr(0,00)) = Box(board(B1)),Box(board(B2))
(1,0):Box(board(B1),addr(1,00)) = Box(board(B0)),Box(board(B3))
(0,1):Box(board(B2),addr(0,01)) = Box(board(B0)),Box(board(B3)),Box(board(B4))
(1,1):Box(board(B3),addr(1,01)) = Box(board(B1)),Box(board(B2)),Box(board(B5))
(0,2):Box(board(B4),addr(0,10)) = Box(board(B2)),Box(board(B5))
(1,2):Box(board(B5),addr(1,10)) = Box(board(B3)),Box(board(B4))
+board_board_cost  = 8                    // <!> Relative to board::mailbox_cost

[box(CommonBox)]                          // Definition of box type "CommonBox"
+box_board_cost    = *                    // <!> Missing
+supervisor_memory = 10240                // Coleridge: 46GB RAM, here reserved 10GB

[board(CommonBoard)]                      // Definition of board type "CommonBoard"
//   0 -- 1 -- 2 -- 3
//   |    |    |    |
//   4 -- 5 -- 6 -- 7
//   |    |    |    |
//   8 -- 9 -- A -- B                     // Mailbox *type* not specified here because
//   |    |    |    |                     // the instance picks up the default type
//   C -- D -- E -- F                     // (here "CommonMbox")
(0,0):Mbox0(addr(00,00)) = Mbox1,Mbox4
(1,0):Mbox1(addr(01,00)) = Mbox0,Mbox2,Mbox5
(2,0):Mbox2(addr(10,00)) = Mbox1,Mbox3,Mbox6
(3,0):Mbox3(addr(11,00)) = Mbox2,Mbox7
.
.
(1,3):MboxD(addr(01,11)) = Mbox9,MboxC,MboxE
(2,3):MboxE(addr(10,11)) = MboxA,MboxD,MboxF
(3,3):MboxF(addr(11,11)) = MboxB,MboxE
+board_mailbox_cost   = *                 // <!> Missing
+supervisor_memory    = 0
+mailbox_mailbox_cost = 1                 // <!> Relative to box::board_board_cost
+dram                 = 4096              // MiB, two DDR3 DRAM boards.

[mailbox(CommonMbox)]                     // Definition of "CommonMbox"
+cores                = 4                 // Cores on mailbox (default = "CommonCore")
+mailbox_core_cost    = *                 // <!> Missing
+core_core_cost       = *                 // <!> Missing

[core(CommonCore)]                        // Definition of "CommonCore"
+threads              = 16                // Threads on core - threads are typeless
+instruction_memory   = 8                 // KiB
+data_memory          = *                 // <!> Missing
+core_thread_cost     = *                 // <!> Missing
+thread_thread_cost   = *                 // <!> Missing
```

Figure III.4.6.4.a: Hardware definition file



```
[engine_box]
Box(boards(B0,B1,B2,B3,B4,B5))
+external_box_cost = *
[engine_board]
(0,0):Box(board(B0),addr(0,00)) = \
      Box(board(B1)),Box(board(B2))
.
.
+board_board_cost  = 8
[box(CommonBox)]
+box_board_cost    = *
+supervisor_memory = 10240
[board(CommonBoard)]
(3,2):MboxB(addr(11,10)) = \
      Mbox7,MboxA,MboxF
.
.
+board_mailbox_cost   = *
+supervisor_memory    = 0
+mailbox_mailbox_cost = 1
+dram                 = 4096
[mailbox(CommonMbox)]
+cores                = 4
+mailbox_core_cost    = *
+core_core_cost       = *
[core(CommonCore)]
+threads              = 16
+instruction_memory   = 8
+data_memory          = *
+core_thread_cost     = *
+thread_thread_cost   = *
```

Figure III.4.6.4.b: Hardware configuration

4.6.5 XML grammar definition **`V4Grammar2.ocfg`**

The POETS XML grammar definition is itself defined in XML. Currently, it looks something like figure III.4.6.5.a. Section 4.6.1 supplies the exact location of the current file.

```
<?xml version="1.0"?>
<!-- V4 Grammar definition file, 2020-04-18, Gospel according to Mark. -->
<Graphs Graphs="[1],[1]xmlns,[0..1]formatMinorVersion,[1]appname">
  <GraphType GraphType="[1],[1]id">
    <Properties Properties="[0..1]"><CDATA CDATA="[1]"/></Properties>
    <MetaData MetaData="[]"/>
    <SharedCode SharedCode="[0..1]"><CDATA CDATA="[1]"/></SharedCode>
    <MessageTypes MessageTypes="[0..1]">
      <MessageType MessageType="[],[1]id"><CDATA CDATA="[1]"/></MessageType>
    </MessageTypes>
    <DeviceTypes DeviceTypes="[1]">
      <DeviceType DeviceType="[],[1]id">
        <Properties Properties="[0..1]"><CDATA CDATA="[1]"/></Properties>
        <State State="[0..1]"><CDATA CDATA="[1]"/></State>
        <SharedCode SharedCode="[0..1]"><CDATA CDATA="[1]"/></SharedCode>
        <SupervisorOutPin SupervisorOutPin="[0..1]">
          <OnSend OnSend="[1]"><CDATA CDATA="[1]"/></OnSend>
        </SupervisorOutPin>
        <SupervisorInPin SupervisorInPin="[0..1]">
          <OnReceive OnReceive="[1]"><CDATA CDATA="[1]"/></OnReceive>
        </SupervisorInPin>
        <InputPin InputPin="[],[1](name,messageTypeId)">
          <Properties Properties="[0..1]"><CDATA CDATA="[1]"/></Properties>
          <State State="[0..1]"><CDATA CDATA="[1]"/></State>
          <OnReceive OnReceive="[1]"><CDATA CDATA="[1]"/></OnReceive>
        </InputPin>
        <OutputPin OutputPin="[],[1](name,messageTypeId)">
          <OnSend OnSend="[1]"><CDATA CDATA="[1]"/></OnSend>
        </OutputPin>
        <ReadyToSend ReadyToSend="[0..1]"><CDATA CDATA="[1]"/></ReadyToSend>
        <OnInit OnInit="[0..1]"><CDATA CDATA="[1]"/></OnInit>
        <OnHardwareIdle OnHardwareIdle="[0..1]"><CDATA CDATA="[1]"/></OnHardwareIdle>
        <OnDeviceIdle OnDeviceIdle="[0..1]"><CDATA CDATA="[1]"/></OnDeviceIdle>
        <MetaData MetaData="[]"/>
      </DeviceType>
      <ExternalType ExternalType="[],[1]id">
        <Properties Properties="[0..1]"><CDATA CDATA="[1]"/></Properties>
        <InputPin InputPin="[],[1](name,messageTypeId)"/>
        <OutputPin OutputPin="[],[1](name,messageTypeId)"/>
        <MetaData MetaData="[]"/>
      </ExternalType>
      <SupervisorType SupervisorType="[0..1],[1]id,[0..1](SupervisorInPin,SupervisorOutPin)">
        <Code Code="[0..1]"><CDATA CDATA="[1]"/></Code>
        <SupervisorOutPin SupervisorOutPin="[0..1],[1]id,[1]messageTypeId">
          <OnSend OnSend="[1]"><CDATA CDATA="[1]"/></OnSend>
        </SupervisorOutPin>
        <SupervisorInPin SupervisorInPin="[0..1],[1]id,[1]messageTypeId">
          <OnReceive OnReceive="[1]"><CDATA CDATA="[1]"/></OnReceive>
        </SupervisorInPin>
        <OnSupervisorIdle OnSupervisorIdle="[0..1]"><CDATA CDATA="[1]"/></OnSupervisorIdle>
        <OnRTCL OnRTCL="[..1]"><CDATA CDATA="[1]"/></OnRTCL>
        <OnStop OnStop="[..1]"><CDATA CDATA="[1]"/></OnStop>
        <OnCTL  OnCTL="[..1]"><CDATA CDATA="[1]"/></OnCTL>
        <MetaData MetaData="[]"/>
      </SupervisorType>
    </DeviceTypes>
  </GraphType>
  <GraphInstance GraphInstance="[],[1](id,graphTypeId)">
    <MetaData MetaData="[]"/>
    <Properties Properties="[0..1]"><CDATA CDATA="[1]"/></Properties>
    <DeviceInstances DeviceInstances="[1]">
      <DevI DevI="[],[1](id,type),[0..1](P,S)">
        <MetaData MetaData="[]"/>
      </DevI>
      <ExtI ExtI="[],[1](id,type),[0..1]P">
        <MetaData MetaData="[]"/>
      </ExtI>
    </DeviceInstances>
    <EdgeInstances EdgeInstances="[1]">
      <EdgeI EdgeI="[],[1]path(path),[0..1](P,S)">
        <MetaData MetaData="[]"/>
      </EdgeI>
    </EdgeInstances>
  </GraphInstance>
</Graphs>
```

Figure III.4.6.5.a: POETS system definition grammar

It is the only Orchestrator configuration file not defined in UIF.

The entire codebase for XML processing has now been transferred to generics, along with the stand-alone documentation (describing the inner syntax of the element name attributes string).

Except it hasn't.

4.7 Testing

Testing something this big and complicated demands a non-trivial and many-layered approach. Here are outlined the six main testing techniques employed in the Orchestrator development. Alongside this, many of the individual C++ classes making up the system have individual test harnesses (currently ~ 70 of them).

4.7.1 Datastructure scanner

The datastructure scanner does little more than walk the entire datastructure, reporting the lack of fields and suspicious constructs where it finds them. It is invoked from the command line:

```
system /integ
```

which invokes an object **`DS_integ`**, which crawls over everything it can find, complaining.

The scanner itself is probably the most immature component in the Orchestrator; further work is needed.

4.7.2 Inversion testing

This is the most powerful testing technique applicable to the Orchestrator. It is also completely currently vapourware, but when time permits......

The idea behind inversion testing is to look at the behaviour of a system in terms of a fucntion, say, **ffunction**. **Input_data** is supplied, and the system transforms it into **output_data**. A second function is then invoked - call it **ifunction** - that takes as *input* the **output_data**, and transforms this to yet another dataset - call this **test_output**. **ifunction** is *defined* to be the inverse of **ffunction**. We then check for isomorphism between **input_data** and **test_output**. If they are the same, we can be reasonably sure that **ffunction** is behaving correctly. It is highly unlikely that a fault will be manifest in **ffunction** and the exact inverse fault appears in **ifunction**, especially if the two are written by different authors. The dataflow is illustrated in figure III.4.7.2.a.
The principal advantage of this technique is that it can be deployed automatically within the

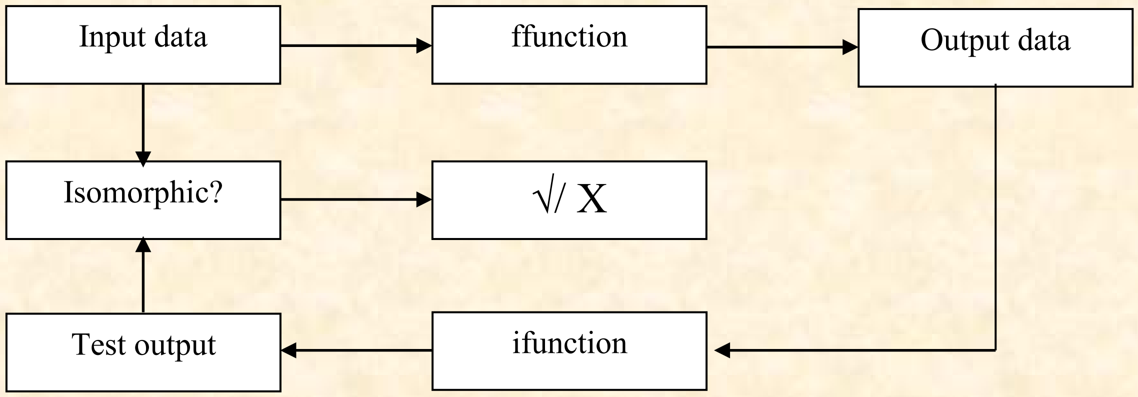


Figure III.4.7.2.a: Inversion testing - the concept

system (at some performance cost) so that every execution provides a different tranche of test data.

The principal disadvantage is the performance hit. Typically **ifunction** will be as expensive as **ffunction**, but often overlooked is the cost of the isomorphism check, which can easily be NP-complete.

In the context of the Orchestrator, the input data is a free-form XML source file, so isomorphism against that is nigh on impossible. However, if we *assume* that the XML-node tree generator is functioning correctly, we can use this as the input data to the test loop. The loop is shortened slightly, but still it tests part of the system: we take the application datastructure, generate from that an XML node tree, and test that against the node tree generated by the Validator. Ignoring the cost of ordering the subnodes at each node, the isomorphism check should be linear.

4.7.3 The Trace subsystem

The Trace subsystem (**`Trace`**) is little more than vapourware, but not much more. The behaviour is that of a fine-granularity machine-readable trajectory of *markers*. It is developer-facing. At strategic intervals throughout the code, strategic variable values are posted to the Trace subsystem, which stores everything in a file. We then create a dedicated browser, that allows the value trajectory of these variables to be displayed in a manner sympathetic to the developer.

4.7.4 Proof-of-life intrinsic applications

The proof-of-life subsystem was installed at the start of the development process, before any real front-end data was available for test. It allows pre-baked (internally formed) applications to be loaded: rings, graphs, trees, cliques, random circuits of all sorts, topologies and sizes. It can provide initial performance data and provide a testing framework. It almost certainly hasn't survived the many re-structurings, and I think it's commented out now.

4.7.5 Dump and Show

These are nothing more than glorified write statements, pretty-printing various sections of the datastructure. The difference is that Dump() is developer-facing (and contains data that is of no interest to the user - for example, pointer values), and Show() is user-facing, presenting the data in a manner intended to be comprehensive to the user.

These capabilities are up-to-date and useful. More or less.

4.7.6 Dummy processes

The Launcher possesses the ability to launch an almost arbitrary number of dummy processes, to stress the MPI network. They are fluid (i.e. they probably don't do anything now) but were very useful in establishing the utility of the asynchronous heterogeneous message-passing capability outlined in section III.2.

4.8 Of packets, internal structure and address layout

Refer to separate documentation "**software_address**".

4.9 User interaction

The Orchestrator internally is a hybrid (multi-thread) heterogeneous MPI universe, that generates binaries and configuration files for the POETS system. It is controlled by the user from a simple, conventional command-line processor.

User *input* is read by a dedicated thread in the Root process.

User *output* comprises two levels: *log* and *microlog* (*ulog* for short). (They are both effectively console writes, but differ in the granularity of detail.)

The coarser of the two, logged output, is timestamped, copied to a log file (by default `....Output\POETS_Logfile.log`), and then repeated via `stdout` to the console. (See section **7: LogServer** process). Logged output consists of high-level action acknowledgements and timings.

The finer (ulog output) is used for parser (error) output, datastructure analysis and dumps: everything that may potentially generate large quantities of output data. This data does not pass through the LogServer and is not archived to the log file. Instead, it can be directed one of three final destinations, controlled by the user. (Note it cannot be switched off.) The destination is controlled with the `path /ulog` command:

Console mode:

```
POETS> path /ulog
```

This sets the ulog stream to `stdout`.

OFGF (one-freebling-great-file) mode:

```
POETS> path /ulog="something\that\resolves\to\a\file.txt"
```

All subsequent ulog output will be sent to the named file.

LOLF (lots-of-little-files) mode:

```
POETS> path /ulog="something/that/resolves/to/a/subdirectory/"
```

... note the closing separator '/' at the end of the string which is what distinguishes a file from a subdirectory.

The output from each separate POETS command will be sent to a unique ulog file in the identified subdirectory. The filename contains the creation time -

```
MicroLog_<YYYY>_<MM>_<DD>_T<HH>_<MM>_<SS>p<n>.plog,
```

where n increments up from 0 if collisions occur. For example:

```
MicroLog_2020_03_11_T16_17_05p0.plog
```

Any new mode takes effect *after* the LogServer has responded to the mode change command, and will remain in effect until overridden; the initial default mode is LOLF.

Both "\" and "/" may be used as directory separators. They can be mixed. If you must. Internally, everything is turned into the Linux '/', because Linux is fussy and Windoze doesn't care.

4.10 Command portfolio (A verb-centric approach)

The user-facing description of the V4 console command set may be found in **Volume IV**. In this section, we describe *only* those commands that involve non-trivial functionality and manipulation of the datastructures.

For the purposes of this explanation, the Orchestrator can be considered to consist of three disjoint substructures. Modulo the internal crosslinks described in section 4.3.4, these substructures are skeletal, in the sense that tearing down the substructures is necessary and sufficient to cleanly destroy the datastructure. The three substructures (described in detail previously) are called **principal structures**. They are

- **The hardware engine model**: This represents the internal Orchestrator model of the target hardware engine in hierarchical form. At the bottom of the hierarchy are threads, which will ultimately be bound to application devices by the cross-linking placement process. There is exactly one hardware engine model in the Orchestrator at any time (although it may be empty).

- **The application side models**: The Orchestrator contains a *set* of **applications** (which may be empty). Each application may contains a *set* of **application graphs** (which may be empty), and a *set* of **type trees** (which may be empty). These two structures implement the **second** and **third** principal structures. On (after) load, there is no connection between them.

Thus the three substructures of interest here are the hardware engine model, the application graphs and the type trees.

The commands outlined below describe a set of actions that may be taken to establish, remove, and inter-link these structures, and to create and deploy the consequent binary images for the POETS hardware. Each command expects to find the datastructure in a defined state, and is guaranteed to leave it in a defined state. Broadly put, if a principal structure exists, it is derived from syntactically and semantically correct input, and is internally consistent. If a load command detects errors, these are reported to the user in a hopefully sympathetic manner and any fragmentary structure deleted. If it's there, it's correct and further user-facing checks unnecessary. (Modulo healthy developer-facing paranoia. Throwing unrecoverable errors at the LogServer is obviously OK.)

Aside from loading and unloading the principal structures, the command set allows the user to perform a number of essential operations on them:

- **Type linking** (tlink): The application *graphs* contain instantiations of named devices, each of which has a named type. These graphs may exist in isolation from any type trees (untypelinked). The application *type trees* contain named definitions of devices, pins and messages. These trees may also exist in isolation from any application graphs (unlinked). The typelinking operation consists of creating links between an application device graph and an application type tree, such that every entity in the graph (notably the graph itself, its constituent pins and devices) are mapped to a corresponding type definition in the type tree. For the typelink to be successful, every entity in the graph must map to an entity in the tree. Multiple application entities may map to a single type definition; it is not necessary for every entity in the tree to be referenced by an application entity. It is possible for a type tree in an application to be

mapped to multiple application graphs, ***both within and outside*** the parent application. A typelink is either successful (in which case the links described above will all exist), or not, in which case no links will persist after the (attempted) operation. A typelink that connects a graph and a tree in the same application is called an **internal** typelink; if the tree and graph are in different applications the typelink may be qualified as **external**.
If typelinks are already in place (and they must, by definition, be correct), they will be deleted before any subsequent typelinking command executes. Thus if a (successful) typelink was in place, and a subsequent typelink fails, the consequence will be that no typelink persists in place.

○ Type unlinking: Typelinks may be explicitly removed in a variety of ways: all the graphs (i.e. everything) in the Orchestrator, all the graphs in a named application, a named graph in a named application, all the graphs mapped to any tree in a named application, all the graphs mapped to a named tree in a named application. Plus anything else I haven't thought of.

Note that untypelinking a graph automatically destroys any pre-existing cross-linking.

- **Placement**: Each application device needs to be mapped to a hardware target thread, under the aegis of the placement subsystem (described elsewhere). Many devices may be - in general, will be - mapped to a single thread (subject to any placement constraints). A placement is either successful (in the sense that all devices are mapped, but the notion of success may be nuanced by the placement subsystem), or not, in which case no cross links will persist. Unlike typelinking, placement is ***not*** commutative: Each placement (attempt) maps an unplaced graph to an engine model that may already contain links to other graphs, so thread capacity issues may perturb the mapping, depending on what's already there.
An application graph must be typelinked before it can be placed.
If a placement fails, for whatever reason, crosslinks to that application graph will be removed; other application mappings will not be affected.

○ Device unplacing: Placement links may be explicitly removed in a variety of ways: all the graphs (i.e. everything) in the Orchestrator, all the graphs in a named application, a named graph in a named application. Plus anything else I haven't thought of.

- Each command is of the form
  command [/*clause* [= *param*[::*param*]^n [, *param*[::*param*]^n]^n]]^n

- All commands and clause names (but not parameters) are truncated to four characters and are case-insensitive. Terms in **bold** are keywords (entered exactly), terms in *italics* are user-defined variable names.

- [+] at the start of a filename (anywhere below) means prepend the appropriate (context-sensitive) path string.

- An *application* is a self-contained XML file that must contain exactly one `Graphs` element.

- Some of the commands provide a finer level of granularity than is generally useful; these are for developer utility.

- We talk glibly about loading and unloading objects and files; for the avoidance of doubt, (input) files are generally thought of as a persistent form of an object, so there is more-or-less a one-to-one correspondence between them.

- Each command handler is a dynamic object, anchored in **`OrchBase`** by the appropriate **`Cm????`** pointer - see figure III.4.1.a. They are all instantiated on initialisation and deleted on closedown, apart from **`CmLoad`**, which is *partly* instantiated on startup and *completely* instantiated on demand.

The workings underlying ***non-obvious*** commands are described below.

Load

Load stuff, mainly from files.

/**app** = [+]*file*

Load an application from a file. If the '+' is present, the application path name defined by "path /apps" is used. The ***first use*** of this command will cause the modules **`XValid`** and **`DS_XML`** to load - figure III.4.10.7.a. They are instantiated from the call to **`CmLoad::operator()`**. It is at *this* point (the first time) that the XML grammar file will be parsed. If any errors are found, they are reported at *this* point.

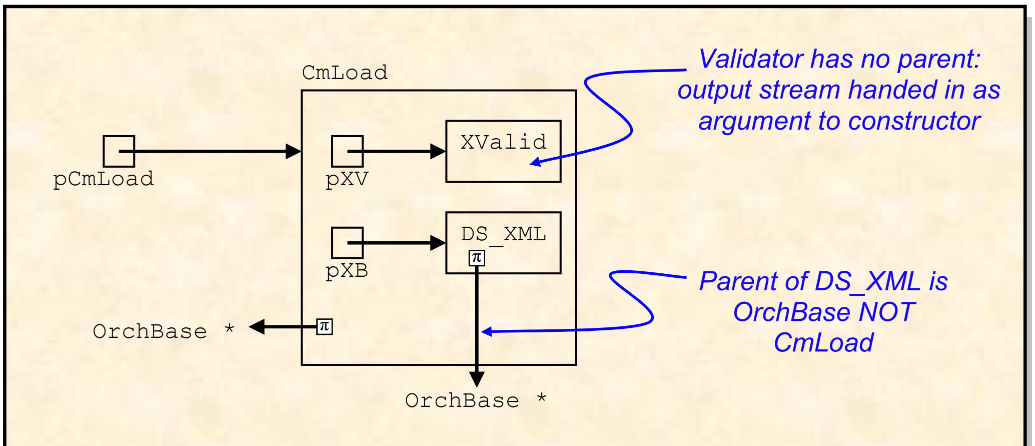

CmLoad
pCmLoad
pXV
XValid
pXB
DS_XML
π
OrchBase *
π
OrchBase *
Validator has no parent: output stream handed in as argument to constructor
Parent of DS_XML is OrchBase NOT CmLoad

Figure III.4.10.7.a: Application load command handler

4.11 Building the application binaries

See separate documentation "**composer**"

5 The NameServer process

See separate documentation "**nameserver**"

6 The UserIO process

See separate documentation "**userio**"

7 The LogServer process

The LogServer is a very simple process, intended to keep track of console output for poring over the bones with hindsight. Any process wanting to communicate with the command console may *'Post'* a message. At this point, the message consists of an unsigned message identifier and a vector of actual strings. (Alternative `Post` signatures allow multiple single strings, up to 8 in number.) This is a quick and simple alternative to variable argument lists and variadic templates. See figure III.7.a.

All messages received by the LogServer are *elaborated*: the process looks up a message format string (indicated by the message id), customises it with the actual strings passed in, and forwards it to the rank 0 console along with a timestamp and a severity (see next section) type. The elaborated timestamped message is also copied to a log file.

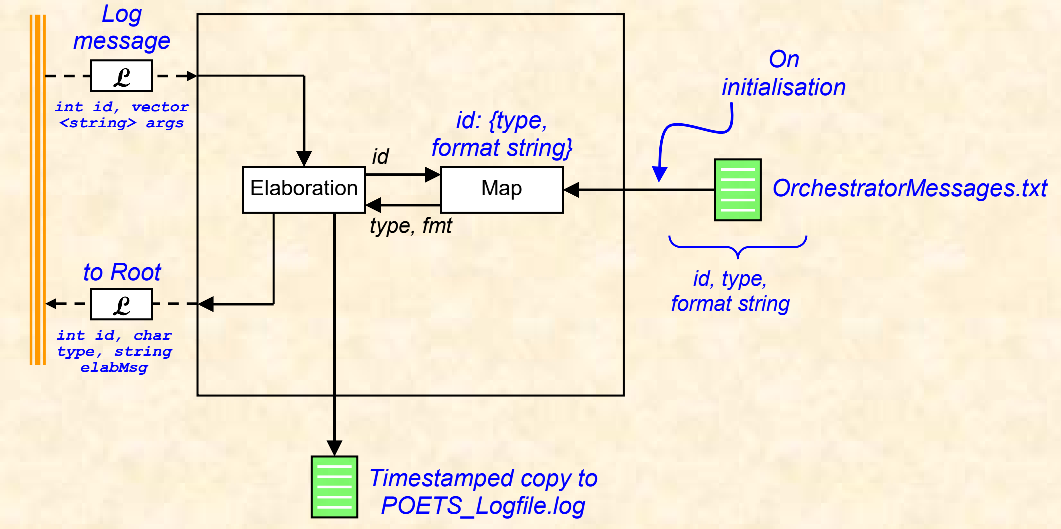

Log message
£
int id, vector <string> args
Elaboration
id
id: {type, format string}
Map
type, fmt
On initialisation
OrchestratorMessages.txt
id, type, format string
to Root
£
int id, char type, string elabMsg
Timestamped copy to POETS_Logfile.log

Figure III.7.a: LogServer internal layout

7.1 Message types

Messages fall into one of eight severity categories. Every message is copied to both the user console and timestamped into the log file. Each is identified by a single character:

(I)nformation: Processing will continue; there is no reason to think that the output will not be correct.

(W)arning: Processing will continue; some rudimentary and trivial corrective action has occurred, but the output is probably correct.

(E)rror: Processing will continue; some corrective action has occurred which will allow the generation of a valid output, but it may not be exactly what the user intended.

(S)evere: Processing will continue; corrective action has been taken, but the final output (if any) should not be trusted. Subsequent errors may arise as a consequence of whatever caused this.

(F)atal: POETS cannot sensibly continue; a diagnostic dump is placed in the log file and the system will attempt to terminate gracefully.

(U)nrecoverable: An unexpected programming error has been detected. Execution of any sort after this message is unreliable; the message itself may be corrupt. The system will attempt to close down, but may not succeed: orphan and/or immortal processes/threads may persist.

(X)enolithic A message sourced from an unknown process. Unlikely to occur in POETS; it is a legacy category.

(D)ebug Development messages that can be switched on/off.

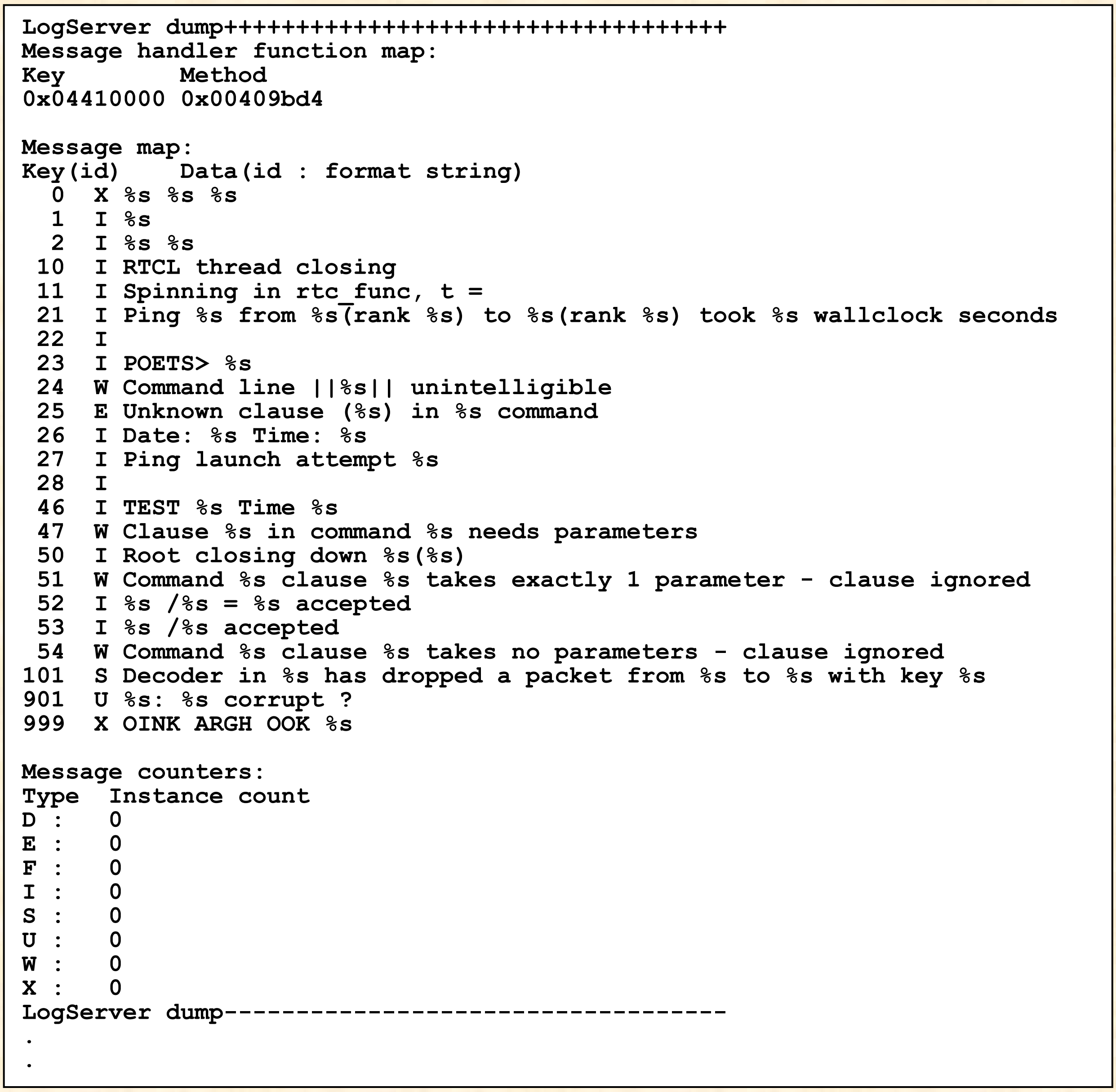
7.2 LogServer message handlers

OnLogP:
Handles the incoming abbreviated message.

OnIdle:
The LogServer is responsible for placing a complete process map into the log file. However, the processes are asynchronous, and although each provides (to all its peers) an environment message, these can arrive in any order, and the LogServer needs to know when it has all the records (i.e. its process map is complete) before writing the map to the log file. LogServer::OnIdle counts the number of environment records it has seen; it knows *a priori* the size of the Universe, hence it knows when the last record arrives and the log entry may be made. It contains a static flag to stop multiple entries.

7.3 LogServer::Dump()

A pretty-print of something not unlike the data structure state is given in figure III.7.3.a.



```
LogServer dump+++++++++++++++++++++++++++++++++++
Message handler function map:
Key        Method
0x04410000 0x00409bd4

Message map:
Key(id)    Data(id : format string)
  0  X %s %s %s
  1  I %s
  2  I %s %s
 10  I RTCL thread closing
 11  I Spinning in rtc_func, t =
 21  I Ping %s from %s(rank %s) to %s(rank %s) took %s wallclock seconds
 22  I
 23  I POETS> %s
 24  W Command line ||%s|| unintelligible
 25  E Unknown clause (%s) in %s command
 26  I Date: %s Time: %s
 27  I Ping launch attempt %s
 28  I
 46  I TEST %s Time %s
 47  W Clause %s in command %s needs parameters
 50  I Root closing down %s(%s)
 51  W Command %s clause %s takes exactly 1 parameter - clause ignored
 52  I %s /%s = %s accepted
 53  I %s /%s accepted
 54  W Command %s clause %s takes no parameters - clause ignored
101  S Decoder in %s has dropped a packet from %s to %s with key %s
901  U %s: %s corrupt ?
999  X OINK ARGH OOK %s

Message counters:
Type  Instance count
D :    0
E :    0
F :    0
I :    0
S :    0
U :    0
W :    0
X :    0
LogServer dump-----------------------------------
.
.
```

Figure III.7.3.a: LogServer::Dump pretty print

8 The real-time clock (RTCL) process

The real time clock process will ultimately support the injection of real-time signals into the POETS graph, under the aegis of some sort of stimulation generation language (as with SpiNNaker).

It consist of two threads, one spinning on the MPI real-time clock, the other handling commands as delivered from the MPI Universe - see figure III.8.a. The spinner thread is fast and its interactions with the communication pool are fixed and regular, and specifically *not* a function of anything the user may be doing, so any time-shear is minimised. Communication between the two threads is via a **communication pool**. The pool is (designed to be) race-free, but this component of the Orchestrator is - subtle....

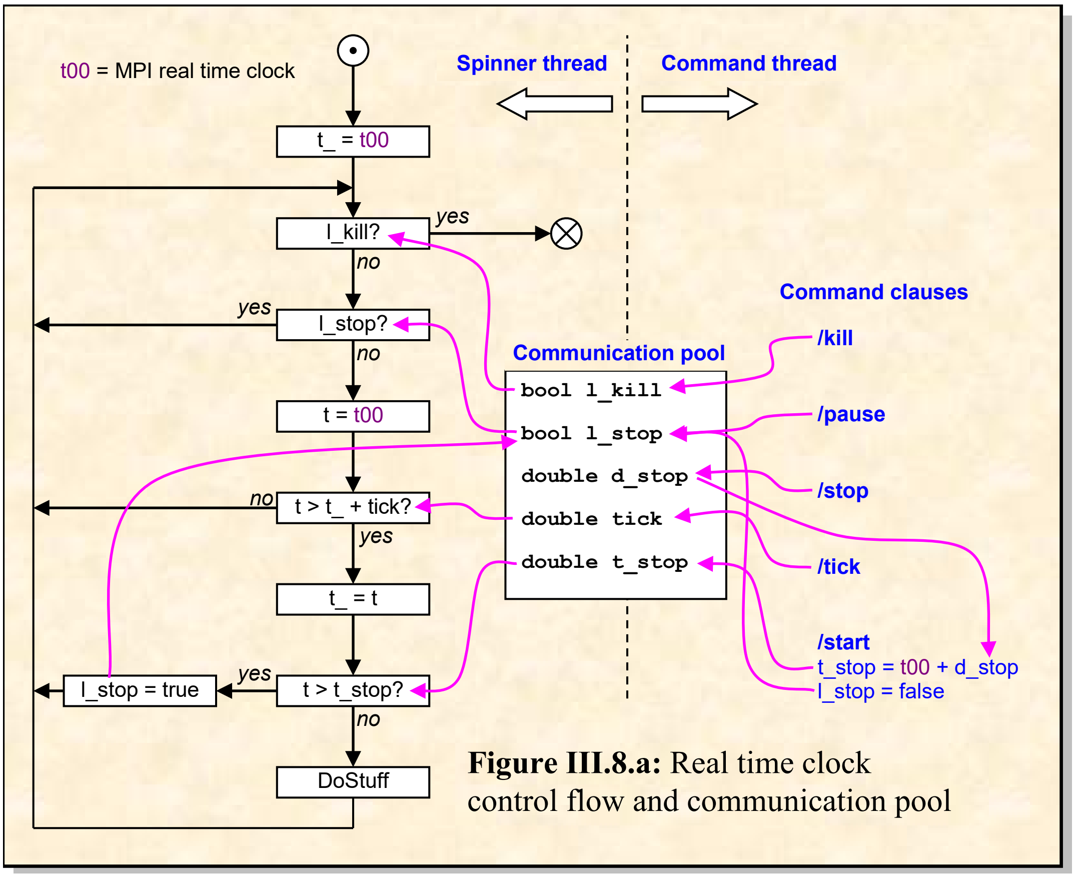


Figure III.8.a: Real time clock control flow and communication pool

8.1 Message handlers

OnExit:
An explicit virtual instance overrides **`CommonBase::OnExit()`** to close down the spinner thread.

OnRTCL:
This reparses the command string and loads the communication pool fields accordingly. For the simplest demonstration of use, type

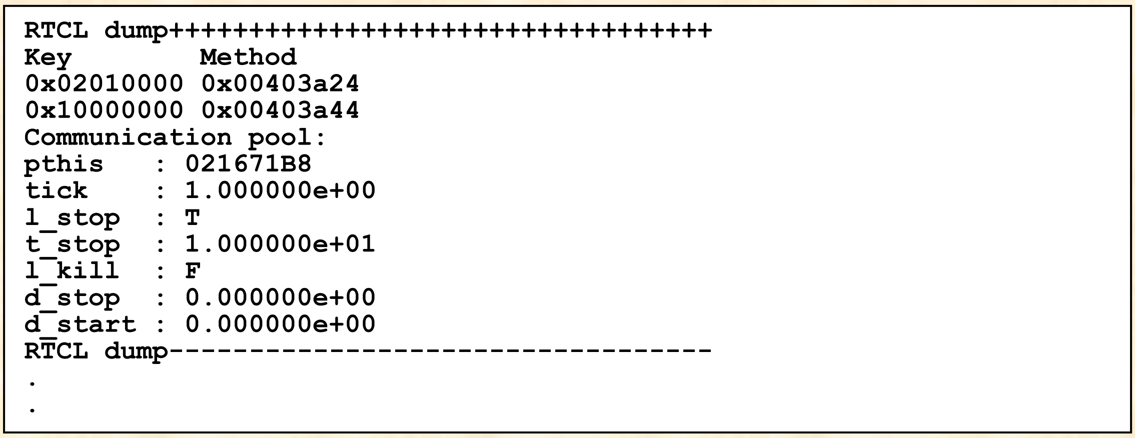
```
POETS> rtcl /stop = 25
POETS> rtcl /start
```

This sets the stop time 25secs after and start command, then starts the clock. The console will display something like

```
.
.
[0]POETS> 13:58:04.83:  11(I) Spinning in rtc_func, t = ..8.000004e+00
[0]POETS> 13:58:05.83:  11(I) Spinning in rtc_func, t = ..9.000004e+00
[0]POETS> 13:58:06.83:  11(I) Spinning in rtc_func, t = ..1.000000e+01
[0]POETS> 13:58:07.82:  11(I) Spinning in rtc_func, t = ..1.100000e+01
.
.
```

8.2 RTCL::Dump()

Figure III.8.2.a shows a pretty-print of the RTCL datastructure state.



```
RTCL dump++++++++++++++++++++++++++++++++++
Key        Method
0x02010000 0x00403a24
0x10000000 0x00403a44
Communication pool:
pthis   : 021671B8
tick    : 1.000000e+00
l_stop  : T
t_stop  : 1.000000e+01
l_kill  : F
d_stop  : 0.000000e+00
d_start : 0.000000e+00
RTCL dump----------------------------------
 .
 .
```

Figure III.8.2.a: RTCL::Dump pretty print

9 The Injector process

The Injector subsystem is an MPI process that supports control of the Orchestrator being passed to executing code, as opposed to the keyboard. It is different from the batch command subsystem in that the command sequence is derived from executing code in the Injector process, rather than a sequence of batch commands. Thus the command flow may be predicated on conditionals.

It is intended for development and debugging; it is not intended to be user-facing.

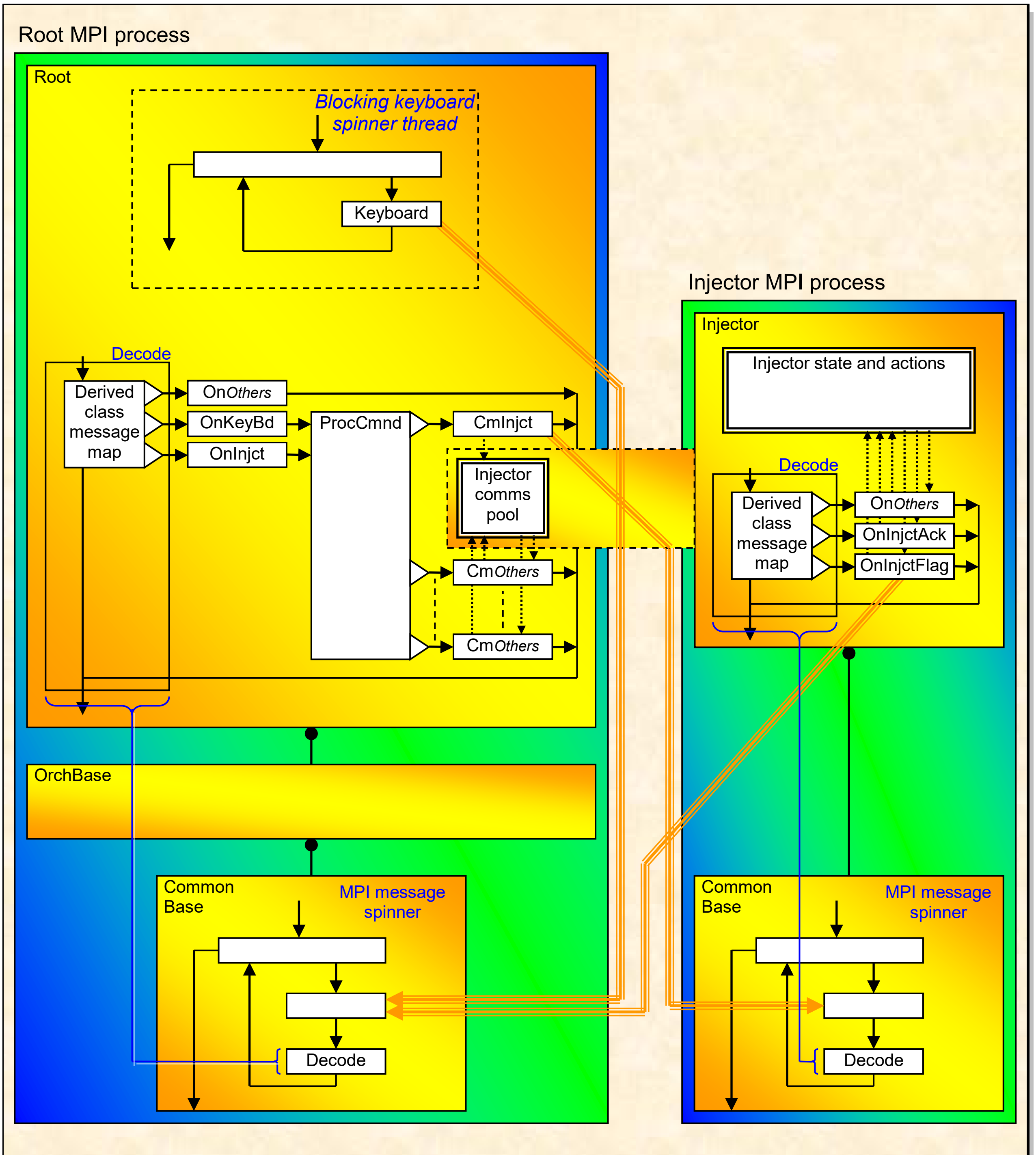


Figure III.9.a: The Injector subsystem

The advantage of this capability is that the control sequence may be arbitrarily complex (it is controlled by executing code, with all the power that that confers). The disadvantages are (1) the code has to be written in the form of an event-driven state transition machine (but then if you're not comfortable with that idea by now you shouldn't be messing with the internals of the Orchestrator anyway), and (2) the Injector has to be recompiled and the system re-launched every time the control sequence is modified. (Encapsulating the Injector behaviour in a class gives code separation, but creates a dependency of the Root class on the Injector class that can trigger automatic rebuilds of the Root when the injector is edited in some IDEs. You can live with it or ignore it in your favourite IDE; rebuilding the Root is not a long process.)

Ignoring the complexities of the base/derived class message maps (described earlier), the structure of the Injector subsystem is shown in figure III.9.a, and the dataflow using it in figure III.9.b.

9.1 Structure

The structure of the Injector subsystem is consistent with the rest of the MPI universe. The injector process consists of a lightweight `main()` that does little but instantiate an Injector class on the heap. `Class Injector` is derived from `CommonBase`, as are all the other principal classes in the Orchestrator universe. `CommonBase` contains the non-blocking MPI message spinner, and the process-specific `Decode` method and function map (`FnMapx`). Two MPI message handlers are provided, `OnInjctAck` and `OnInjctFlag`, described in the next section. Arguably these could have lived in an intermediate class in the inheritance hierarchy (like `OrchBase`), to give the developer completely free reign in the Injector, but there are limits.

9.2 Dataflow

Refer to figure III.9.b. The Injector subsystem is invoked when the user enters an ***INJECT*** command at the keyboard (1). This is caught - as a string - by the thread containing the blocking keyboard read spinner, wrapped in a `PMsg_p` carrier and posted via MPI to itself (process Root). It is picked up by the Root MPI message spinner (2), and passed to

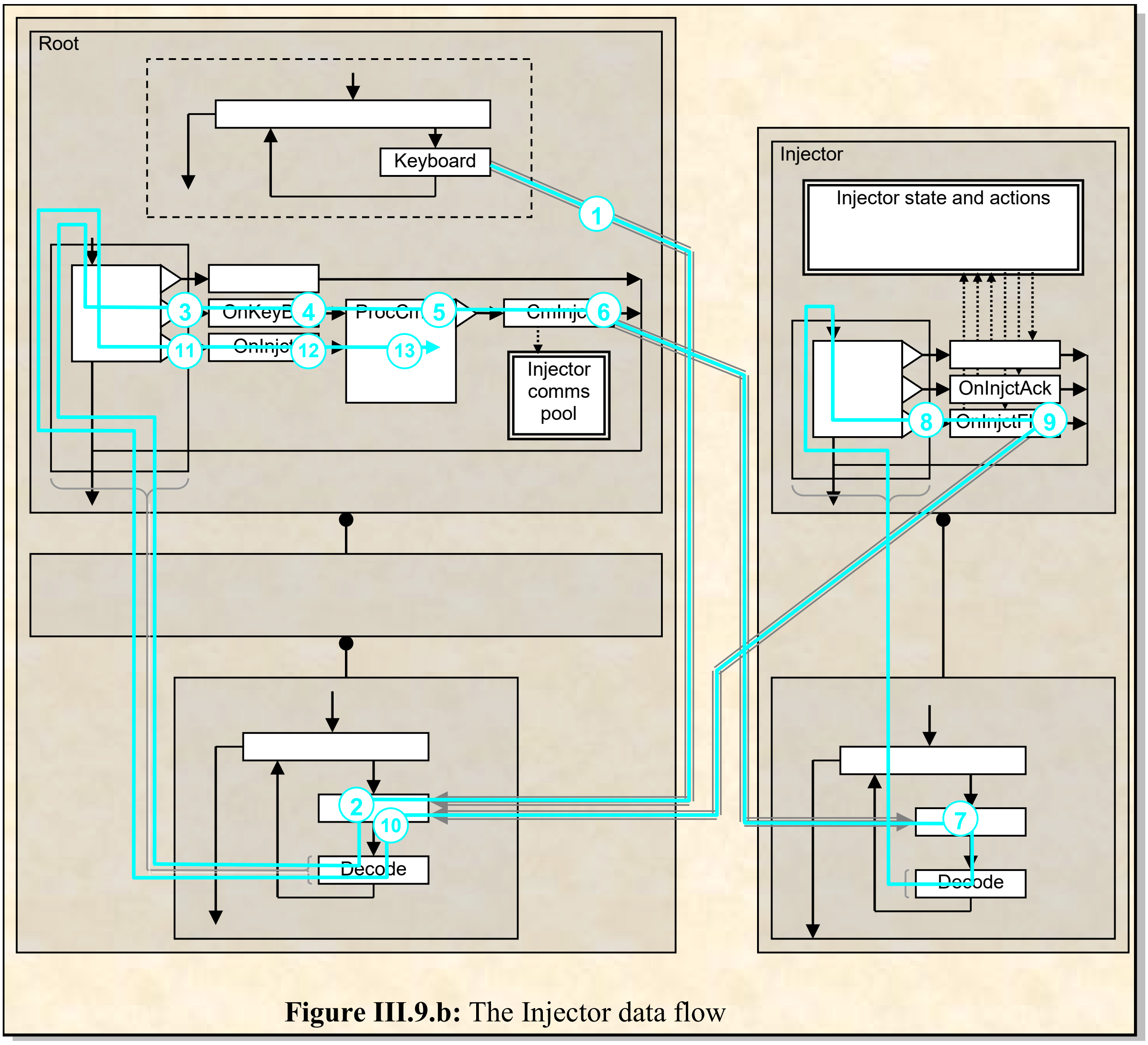


Figure III.9.b: The Injector data flow

Root::Decode (3), where it is recognised as a keyboard string, and turned into a command line object (Cli) (4). From here - as a Cli object - it passes into the **ProcCmnd** switch (5). Here it is recognised and selected as an ***INJECT*** command, and passed to the Root injector command handler **CmInjct** (6). **CmInjct** looks for a ***/FLAG*** clause in the Cli object, and if it finds one, may modulate the data in the Injector communication pool (see next section). That done, the command is streamed back to a **PMsg_p** and sent via MPI to the Injector process, effectively waking it up. (Aside from the startup prologue traffic, nothing is targeted at the Injector up to this point.) The message is picked up by **Injector::MPISpinner** (7) and passed to **Injector::Decode** (8), whence it is passed to the pre-supplied handler **OnInjctFlg** (9).

The rest of the behaviour as shipped is simply for demonstration purposes. Recall that the point of the Injector subsystem is to be able to inject commands into the input stage of the Orchestrator from within program control. **OnInjctFlg** creates a demonstration command string, wraps it in a **P_Msg**, and sends it back to the Root process. It is picked up by **Root::MPIspinner** (10) and **Root::Decode** (11), where it is recognised as a command injected by the Injector, and handed to **Root::OnInjct** (12). From here it is injected into the **Root::ProcCmnd** command switch and handled *in exactly the same way as if it had been typed in at the keyboard.*

9.3 Communication pool

The communication pool is a structure more usually found in shared-memory multi-thread systems (see the section on the real-time clock process control pool). Although there is no actual shared memory between the Root and Injector processes (the Orchestrator emphatically does not use MPI 3 memory windowing), the pool (shipped as a small, uninteresting static structure in the Injector class), enables the developer to behave as if there was.

The structure is instantiated in the **Root** class, giving rise to the dependency mentioned in the introduction. This is a cosmetic decision: the alternative is to make the pool a member of the Root class, which is an unnatural place to house it.

The idea is that the pool can be read/written to pre-and post-action by any of the Root commands, and the structure contents shipped over to the Injector as necessary by any messages sent to it from the Root. Hey, it works.

9.4 Usage

The driver for this functionality is to be able to automate the behaviour of the Orchestrator in ways more complicated than simple batch commands. Specifically, there is a need to make commands injected into the Root dependent on the output of previous commands, which cannot be done in a batch system without effectively inventing another language, with all that that entails. Orchestrator command handlers may be modified to write whatever pre- or post-action state is needed into the pool. The state of the pool can be sent to the Injector explicitly by these commands (data push), or periodically interrogated by the Injector (data pull). The control code in the Injector has to be written in the style of a state transition machine, because obviously the MPI communications traffic to and from the Root is asynchronous.

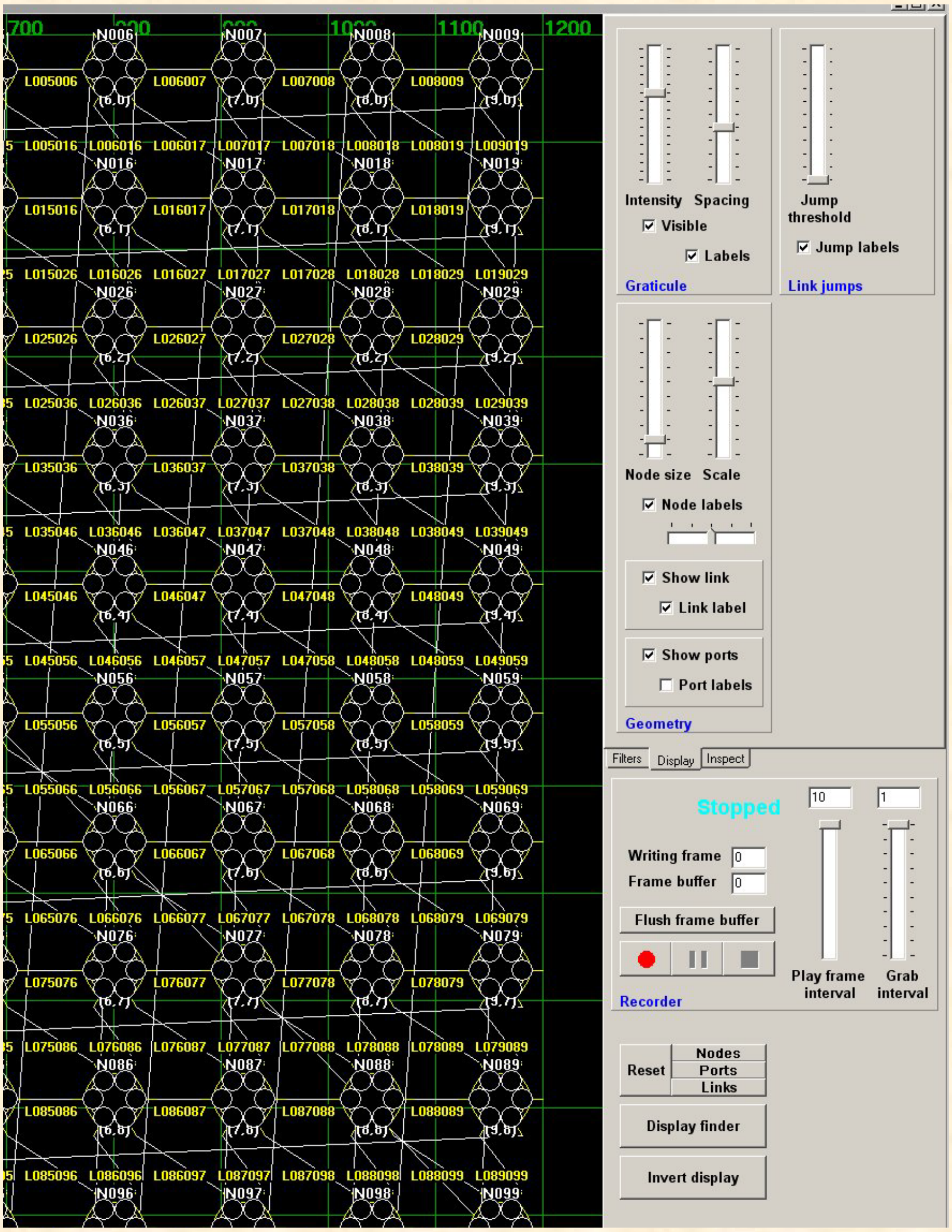
And it all worked, perfectly, first time.

10 The Monitor process

10.1 Function and responsibilities

The idea behind the Monitor processes in general is to allow a remote process, connected to the MPI backplane, to display real-time data about the activities and occupancy of the POETS system. It is largely a cosmetic component (and therefore of low priority), but every supercomputer installation has one in the building reception, so we should too.....

Figure III.10.1.a is nothing more than an indication of the sort of things we could/should do; it's what we did for the SpiNNaker project.


700
1100
1200
N006 N007 N008 N009
L005006 L006007 L007008 L008009
(6,0) (7,0) (8,0) (9,0)
L005016 L006016 L006017 L007017 L007018 L008018 L008019 L009019
N016 N017 N018 N019
L015016 L016017 L017018 L018019
(6,1) (7,1) (8,1) (9,1)
L015026 L016026 L016027 L017027 L017028 L018028 L018029 L019029
N026 N027 N028 N029
L025026 L026027 L027028 L028029
(6,2) (7,2) (8,2) (9,2)
L025036 L026036 L026037 L027037 L027038 L028038 L028039 L029039
N036 N037 N038 N039
L035036 L036037 L037038 L038039
(6,3) (7,3) (8,3) (9,3)
L035046 L036046 L036047 L037047 L037048 L038048 L038049 L039049
N046 N047 N048 N049
L045046 L046047 L047048 L048049
(6,4) (7,4) (8,4) (9,4)
L045056 L046056 L046057 L047057 L047058 L048058 L048059 L049059
N056 N057 N058 N059
L055056 L056057 L057058 L058059
(6,5) (7,5) (8,5) (9,5)
L055066 L056066 L056067 L057067 L057068 L058068 L058069 L059069
N066 N067 N068 N069
L065066 L066067 L067068 L068069
(6,6) (7,6) (8,6) (9,6)
L065076 L066076 L066077 L067077 L067078 L068078 L068079 L069079
N076 N077 N078 N079
L075076 L076077 L077078 L078079
(6,7) (7,7) (8,7) (9,7)
L075086 L076086 L076087 L077087 L077088 L078088 L078089 L079089
N086 N087 N088 N089
L085086 L086087 L087088 L088089
(6,8) (7,8) (8,8) (9,8)
L085096 L086096 L086097 L087097 L087098 L088098 L088099 L089099
N096 N097 N098 N099
Intensity
Spacing
Jump threshold
Graticule
Link jumps
Node size
Scale
Geometry
Filters
Display
Inspect
Flush frame buffer
Play frame interval
Grab interval
Recorder
Reset
Nodes
Ports
Links
Display finder
Invert display

☑ Visible
☑ Labels
☑ Jump labels
☑ Node labels
☑ Show link
☑ Link label
☑ Show ports
☐ Port labels

Stopped
Writing frame 0
Frame buffer 0
Play frame interval 10
Grab interval 1

Figure III.10.1.a: Inspector display of a partial SpiNNaker topology

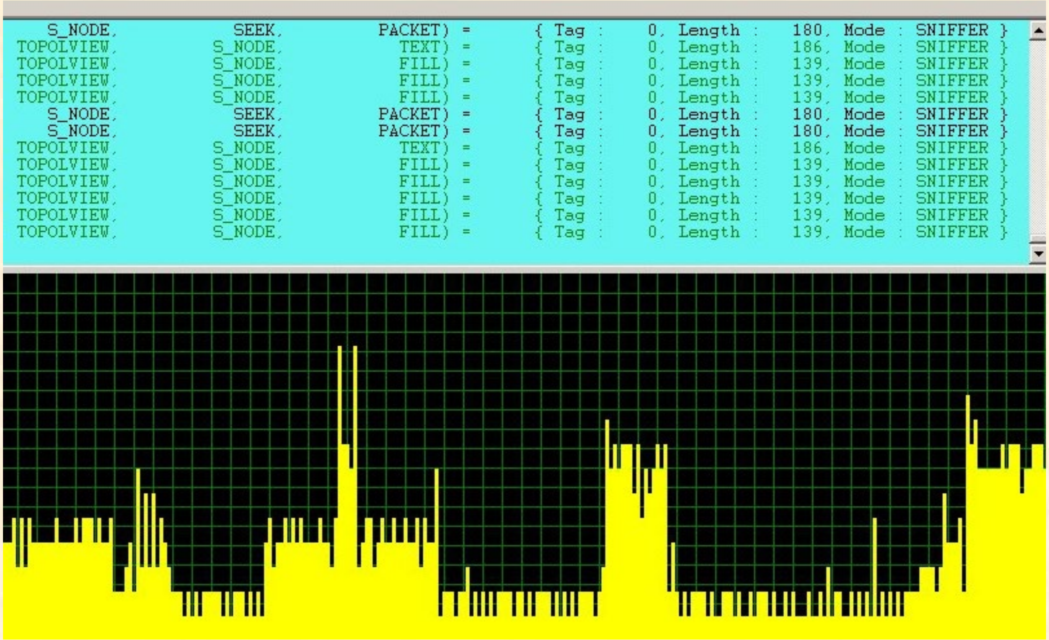
10.2 Real-time inspectors

The real-time inspectors are interactive graphical displays that allow inspection of device states, message fluxes and so on. As with other elements of the system, the original was developed for the SpiNNaker system: figure III.10.1.a shows a partial (SpiNNaker) hardware geometry discovered by the inspector.

Figure III.10.2.a shows an inspector packet-sniffer display.

Both these systems allow the user to drill down to increasing levels of detail.

In POETS, two sources of network data are available: from the MPI network, via the internal instrumentation available through the MPI API, and from POETS, via reporting paths (P_thread .. P_core .. P_node .. Supervisor .. Inspector and so on) that can be enabled from the console.



```
   S_NODE,          SEEK,        PACKET) =     { Tag :    0, Length :    180, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          TEXT) =     { Tag :    0, Length :    186, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          FILL) =     { Tag :    0, Length :    139, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          FILL) =     { Tag :    0, Length :    139, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          FILL) =     { Tag :    0, Length :    139, Mode : SNIFFER }
   S_NODE,          SEEK,        PACKET) =     { Tag :    0, Length :    180, Mode : SNIFFER }
   S_NODE,          SEEK,        PACKET) =     { Tag :    0, Length :    180, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          TEXT) =     { Tag :    0, Length :    186, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          FILL) =     { Tag :    0, Length :    139, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          FILL) =     { Tag :    0, Length :    139, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          FILL) =     { Tag :    0, Length :    139, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          FILL) =     { Tag :    0, Length :    139, Mode : SNIFFER }
TOPOLVIEW,        S_NODE,          FILL) =     { Tag :    0, Length :    139, Mode : SNIFFER }
```

Figure III.10.2.a: SpiNNaker inspector packet sniffer

11 The Dummy processes

11.1 Function and responsibilities

These are development and debug processes that were created to stress the asynchronous communication capabilities of MPI. They play no part in the normal operation of the Orchestrator, and are nominally disabled.

12 Motherships

See separate documentation "**mothership_design**"

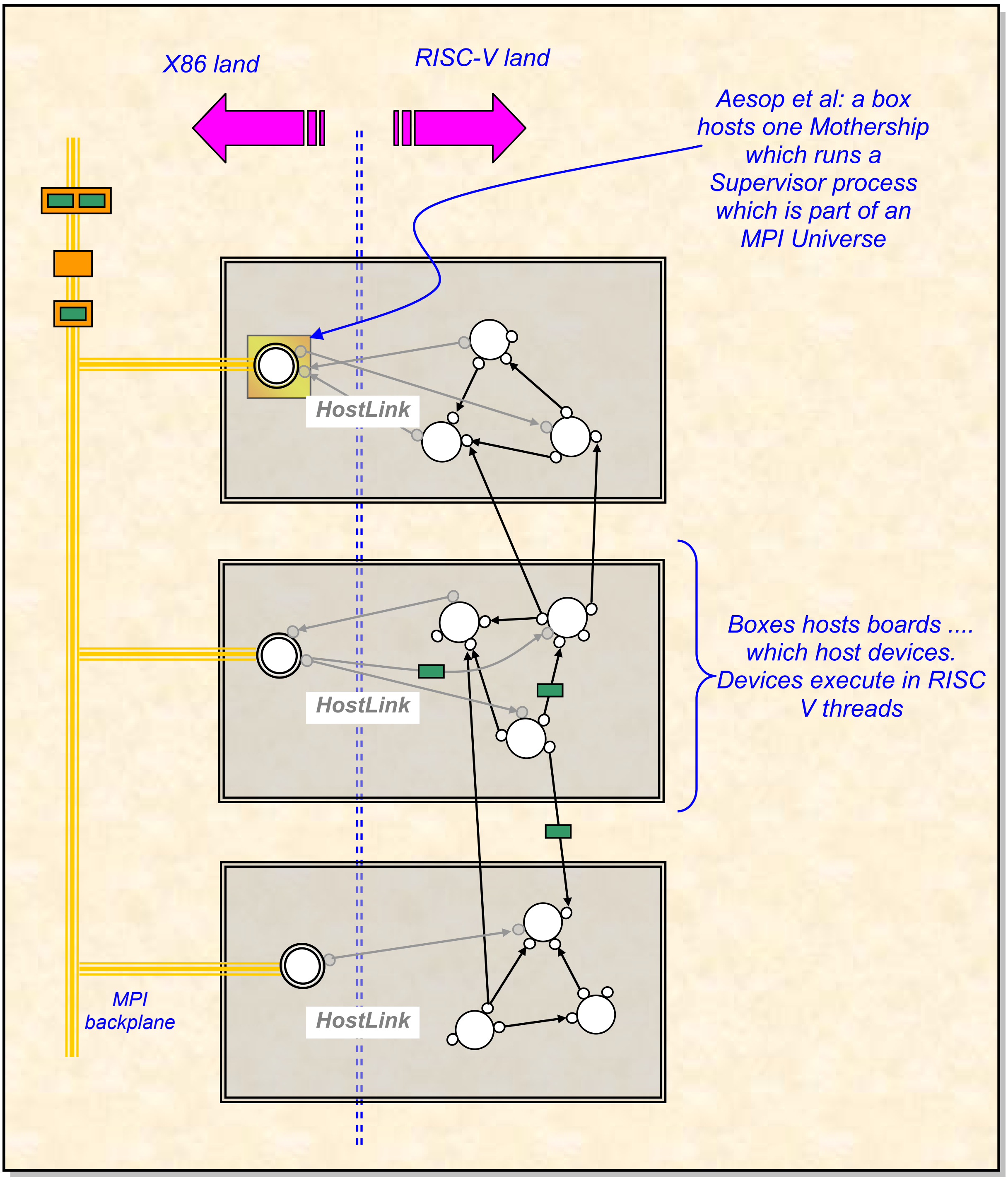
13 Application execution

See separate documentation "**softswitchdocs**"

14 Building and deploying the Orchestrator

See separate documentation "**launcher**".

MISC SPARE COOL PIX


X86 land
RISC-V land
Aesop et al: a box hosts one Mothership which runs a Supervisor process which is part of an MPI Universe
HostLink
HostLink
HostLink
Boxes hosts boards .... which host devices. Devices execute in RISC V threads
MPI backplane

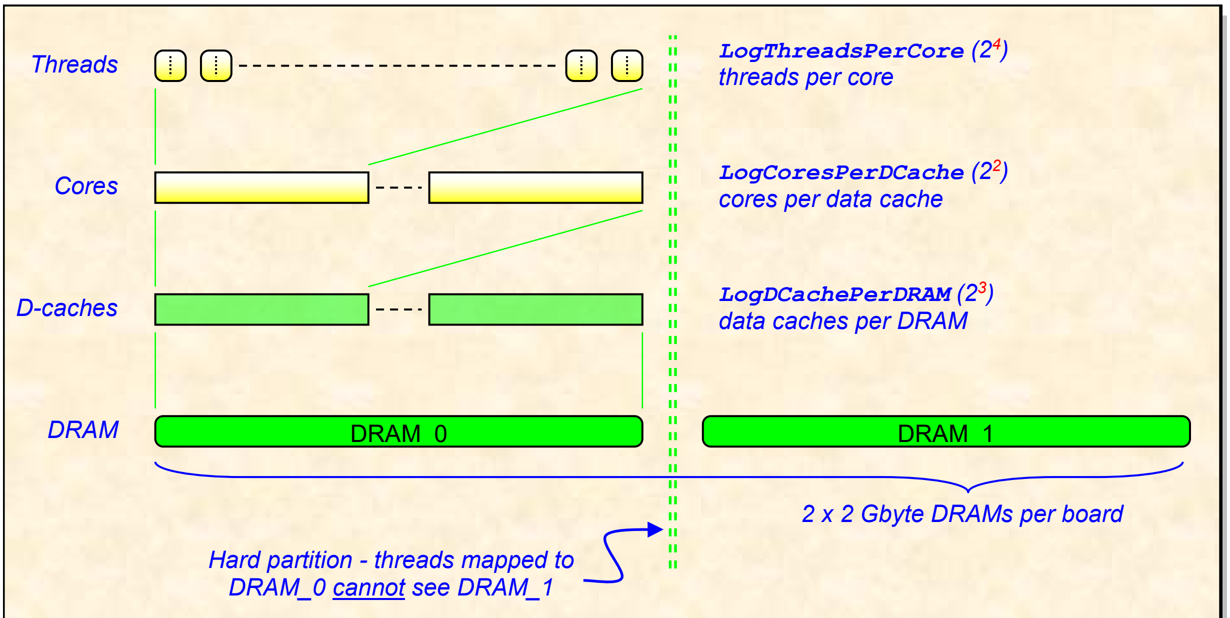


Figure III.4.8.2.a: Application thread physical data memory layout (**AD-space**)

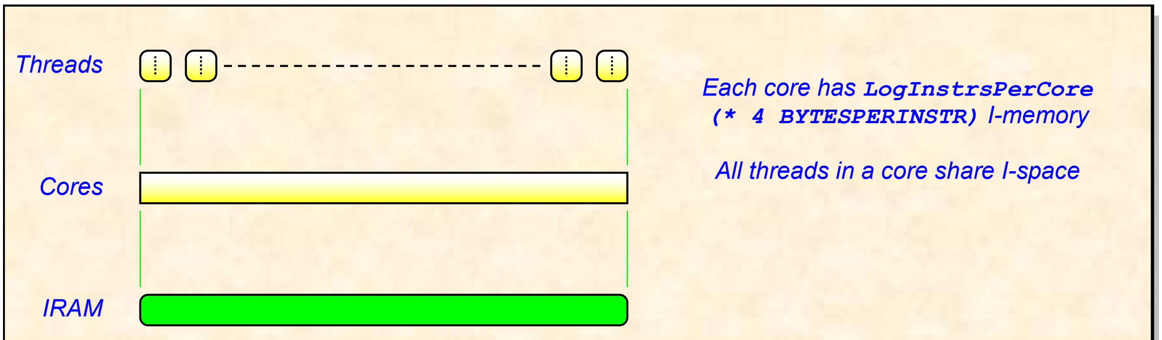

Threads
Cores
IRAM
Each core has LogInstrsPerCore
(* 4 BYTESPERINSTR) I-memory
All threads in a core share I-space

Figure III.4.8.2.b: Application thread physical instruction memory layout (**AI-space**)

Building the binaries

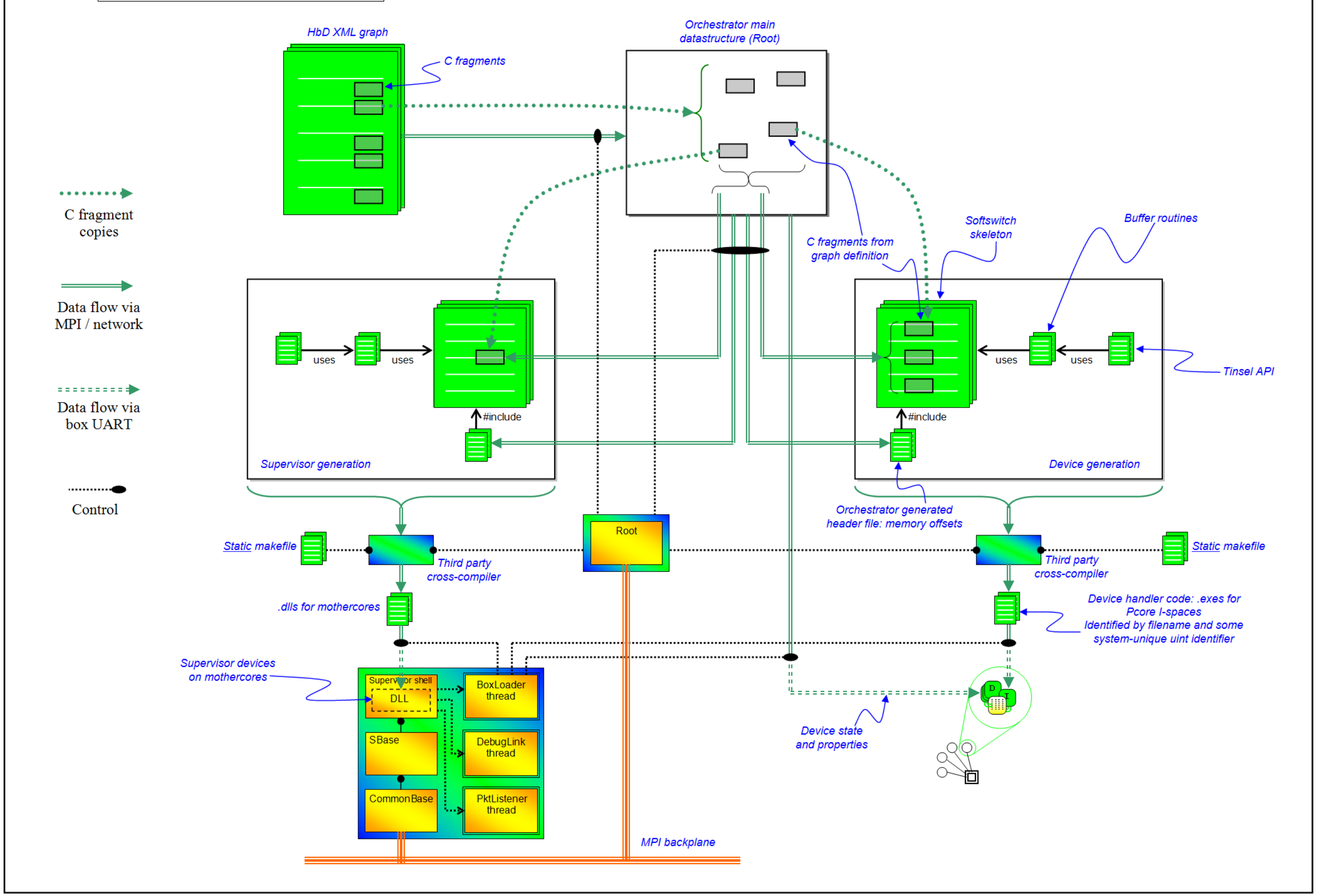


Figure III.4.11.a: Building the binaries

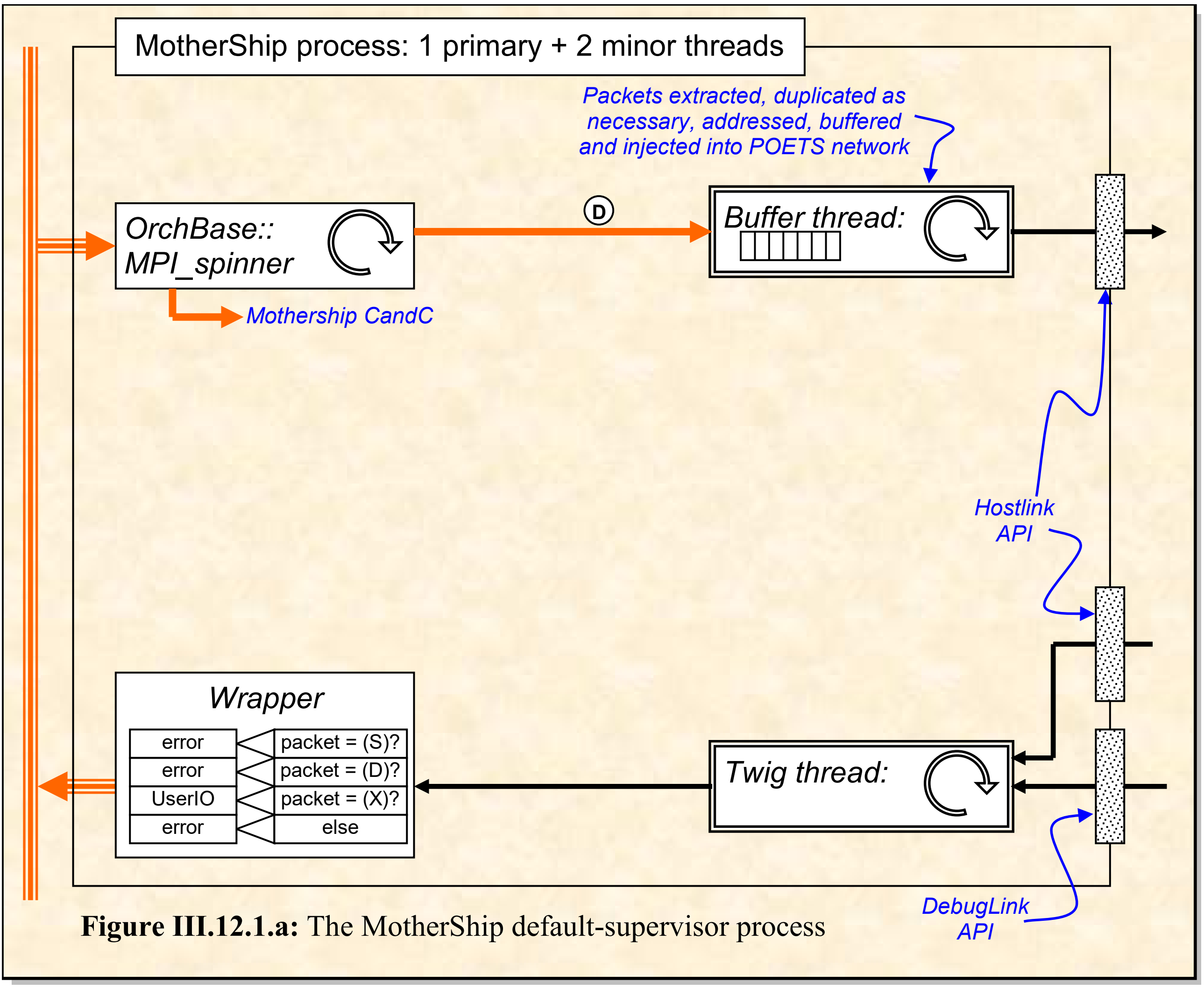
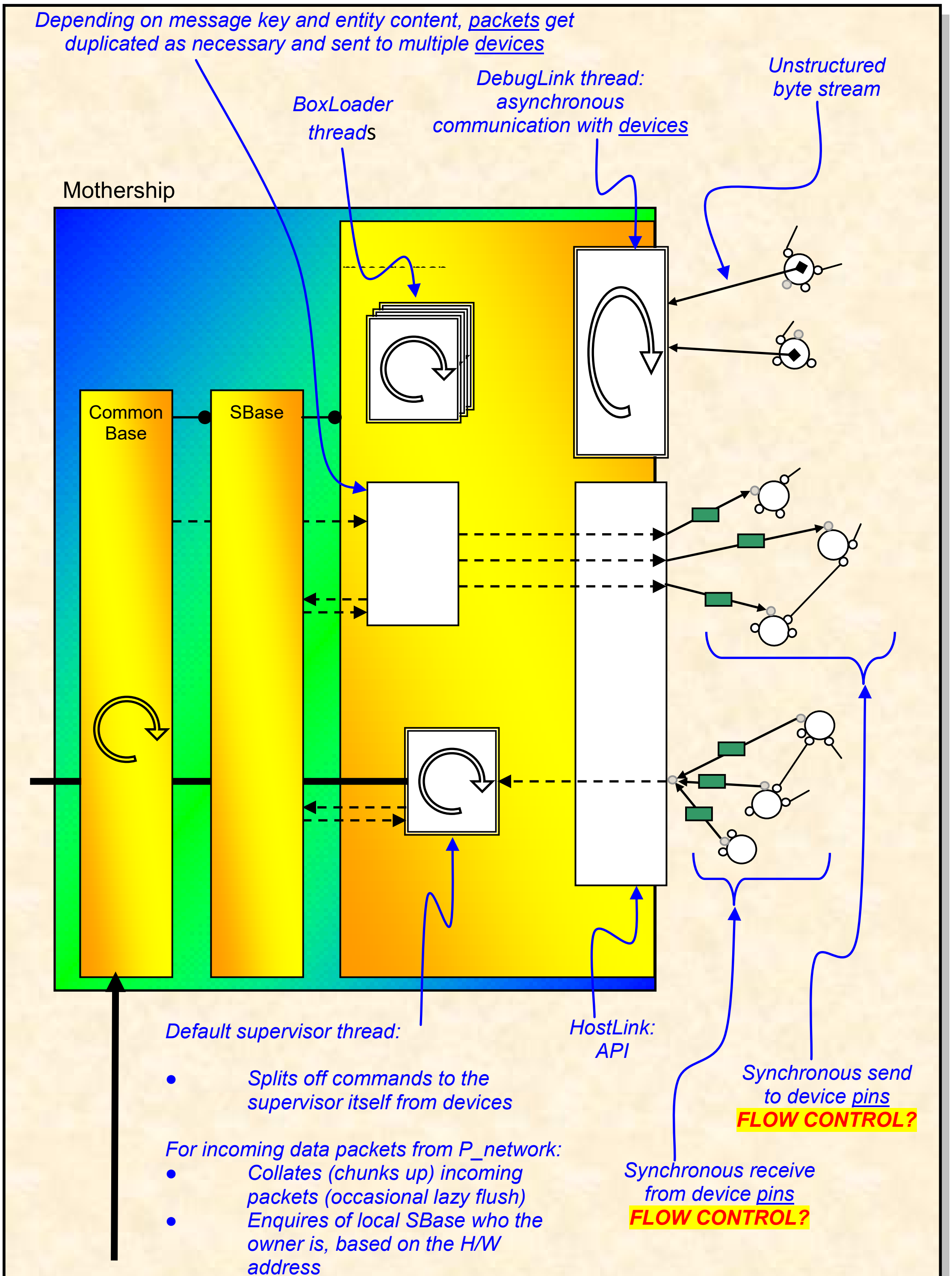


Figure III.12.1.a: The MotherShip default-supervisor process



Depending on message key and entity content, packets get duplicated as necessary and sent to multiple devices

BoxLoader threads

DebugLink thread: asynchronous communication with devices

Unstructured byte stream

Default supervisor thread:

- Splits off commands to the supervisor itself from devices

For incoming data packets from P_network:

- Collates (chunks up) incoming packets (occasional lazy flush)
- Enquires of local SBase who the owner is, based on the H/W address
- Creates an Entity containing the owner
- Sends message to UserIO

HostLink: API

Synchronous send to device pins
FLOW CONTROL?

Synchronous receive from device pins
FLOW CONTROL?

Figure III.12.6.a: Mothership internal structure

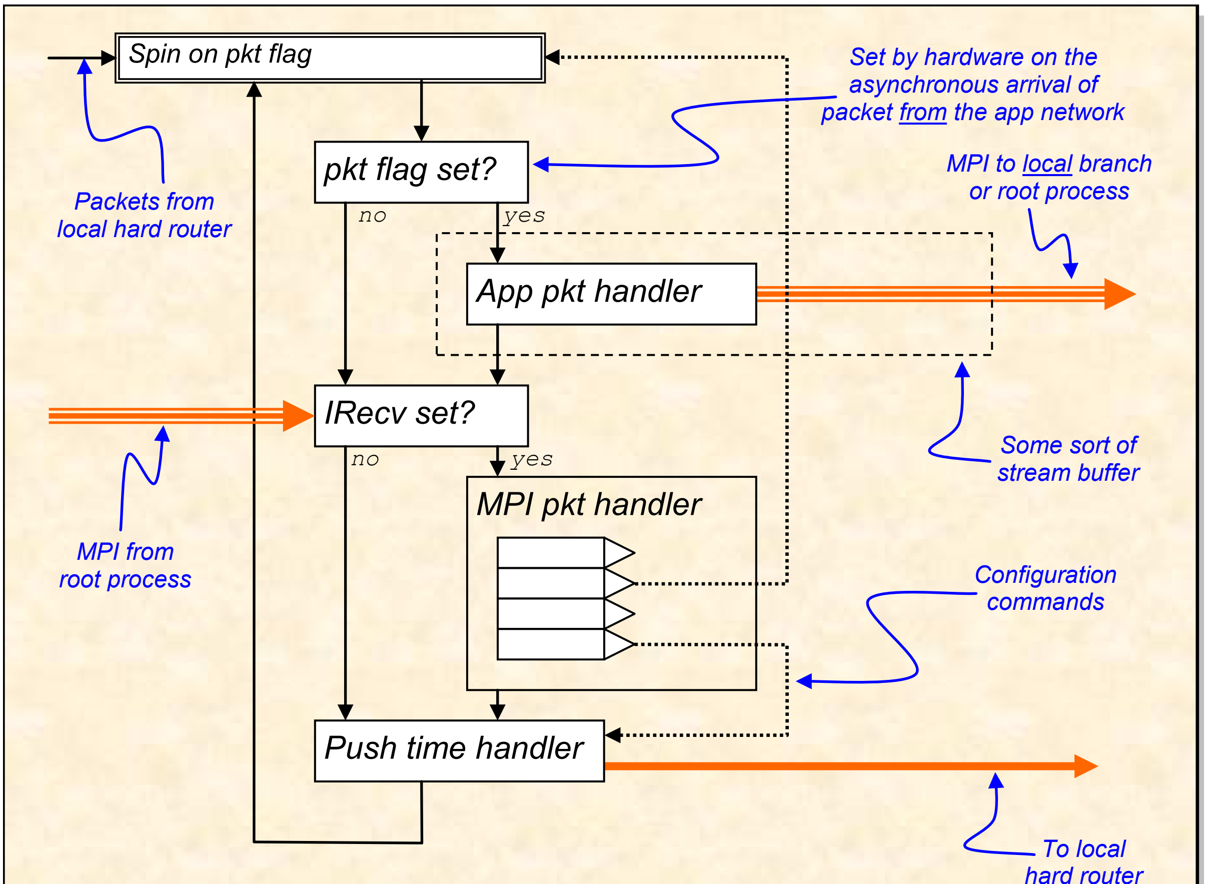

Spin on pkt flag
pkt flag set?
no
yes
App pkt handler
IRecv set?
no
yes
MPI pkt handler
Push time handler
Set by hardware on the asynchronous arrival of packet from the app network
MPI to local branch or root process
Packets from local hard router
Some sort of stream buffer
MPI from root process
Configuration commands
To local hard router

Figure III.4.17: The Twig process

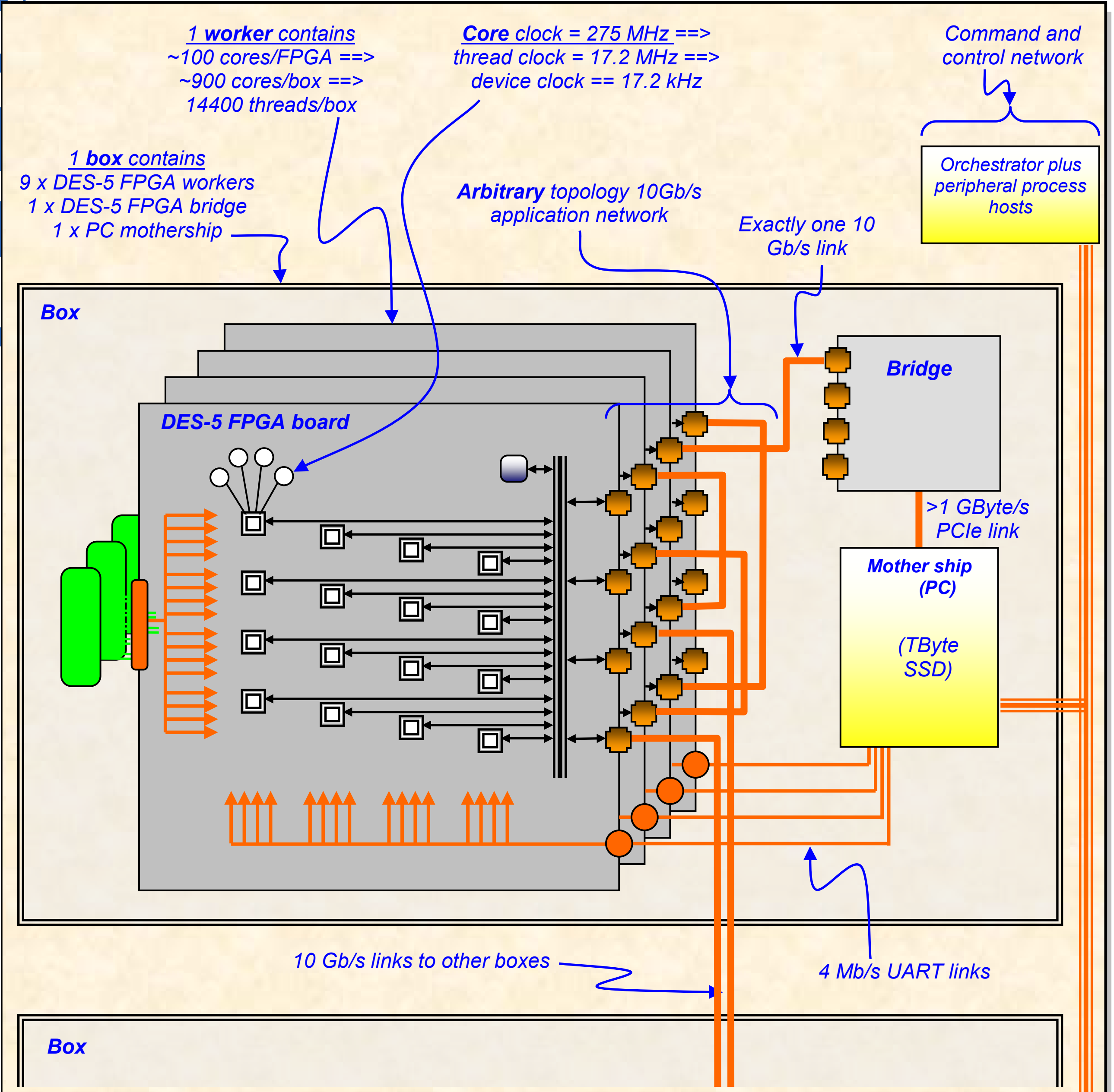


Figure III.12.4: A POETS system prototype

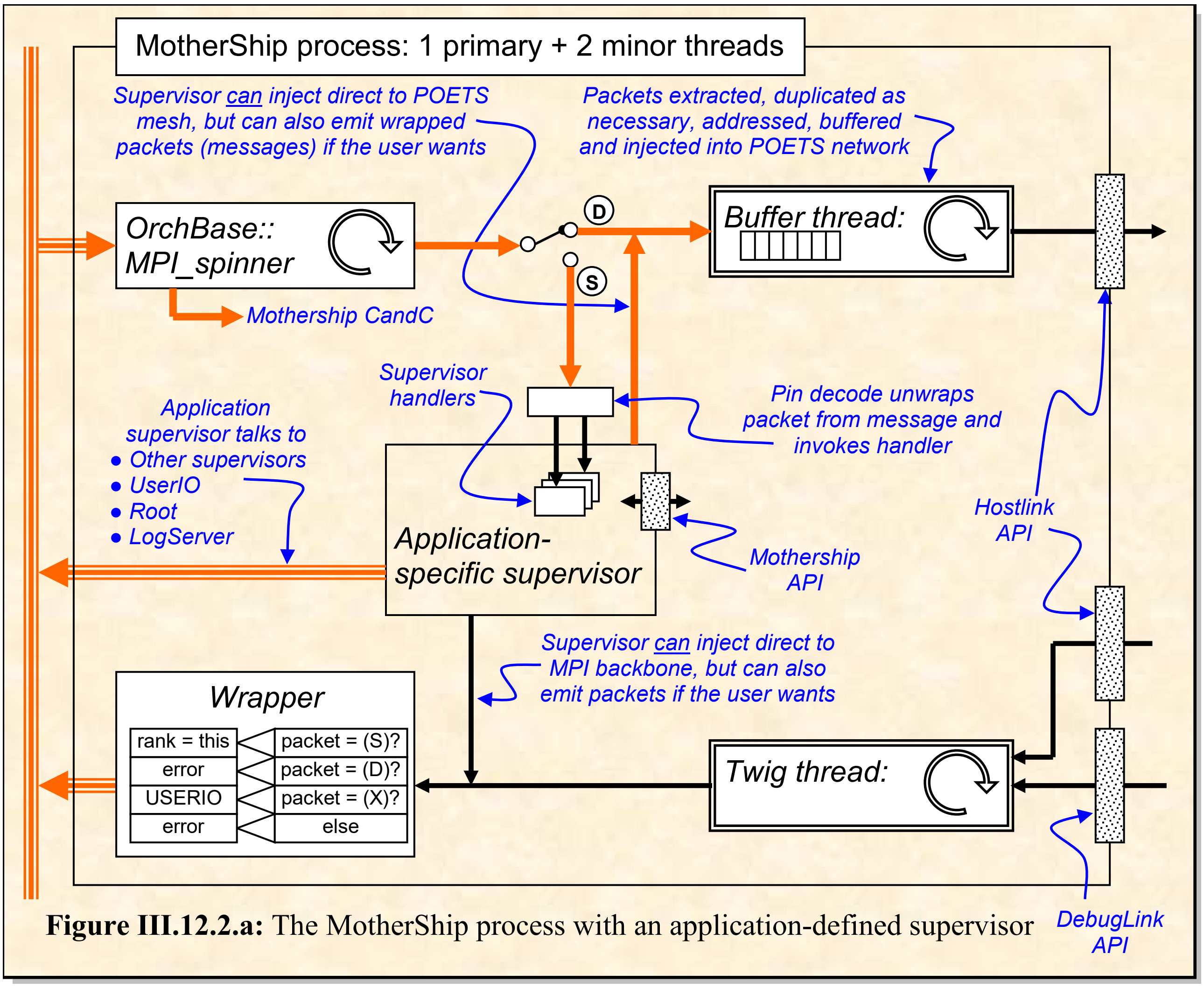

MotherShip process: 1 primary + 2 minor threads
Supervisor can inject direct to POETS mesh, but can also emit wrapped packets (messages) if the user wants
Packets extracted, duplicated as necessary, addressed, buffered and injected into POETS network
OrchBase:: MPI_spinner
D
S
Buffer thread:
Mothership CandC
Supervisor handlers
Application supervisor talks to
● Other supervisors
● UserIO
● Root
● LogServer
Pin decode unwraps packet from message and invokes handler
Application-specific supervisor
Mothership API
Hostlink API
Supervisor can inject direct to MPI backbone, but can also emit packets if the user wants
Wrapper
rank = this
error
USERIO
error
packet = (S)?
packet = (D)?
packet = (X)?
else
Twig thread:
DebugLink API

Figure III.12.2.a: The MotherShip process with an application-defined supervisor

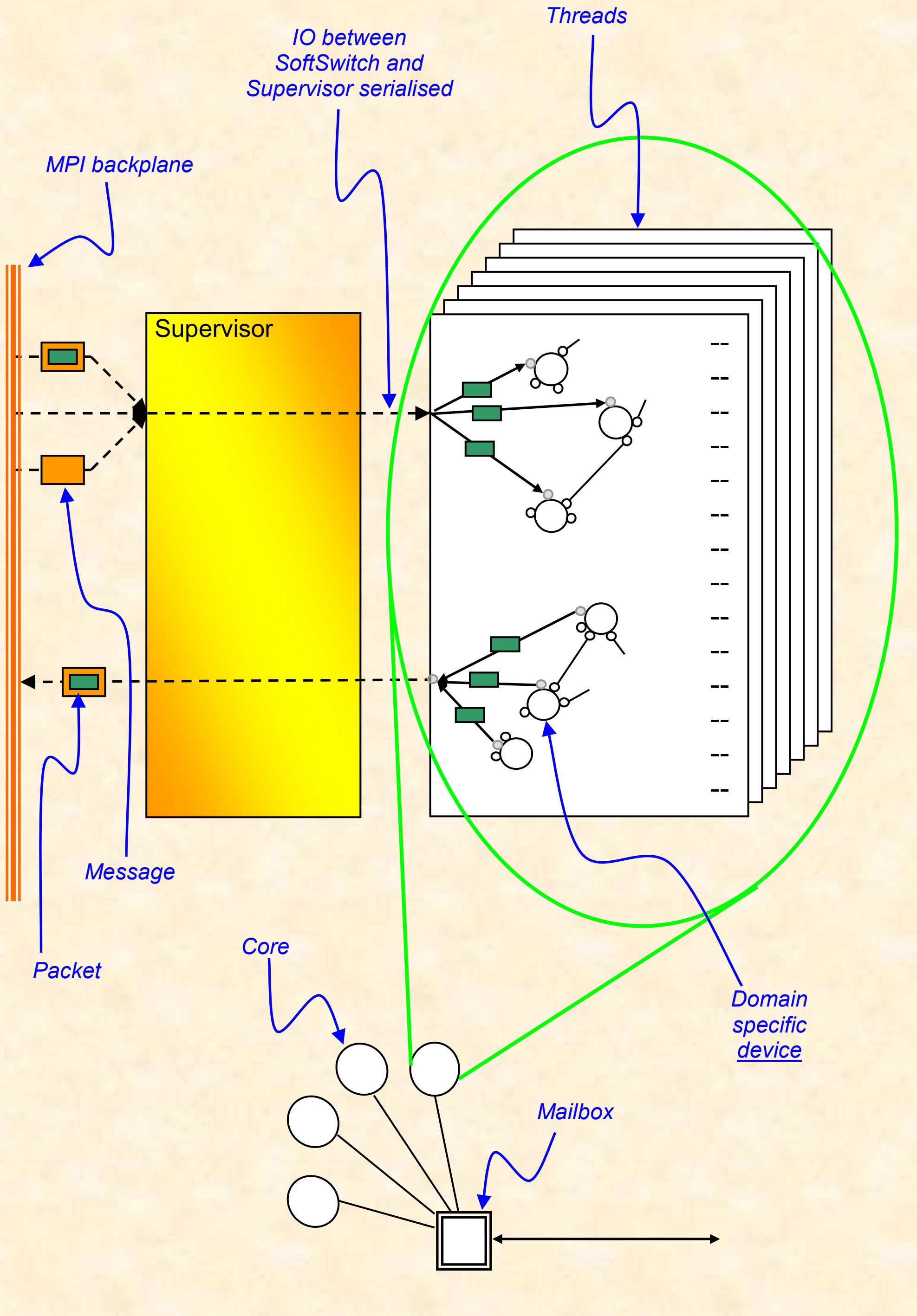


Figure III.12.4: Supervisor/thread communications

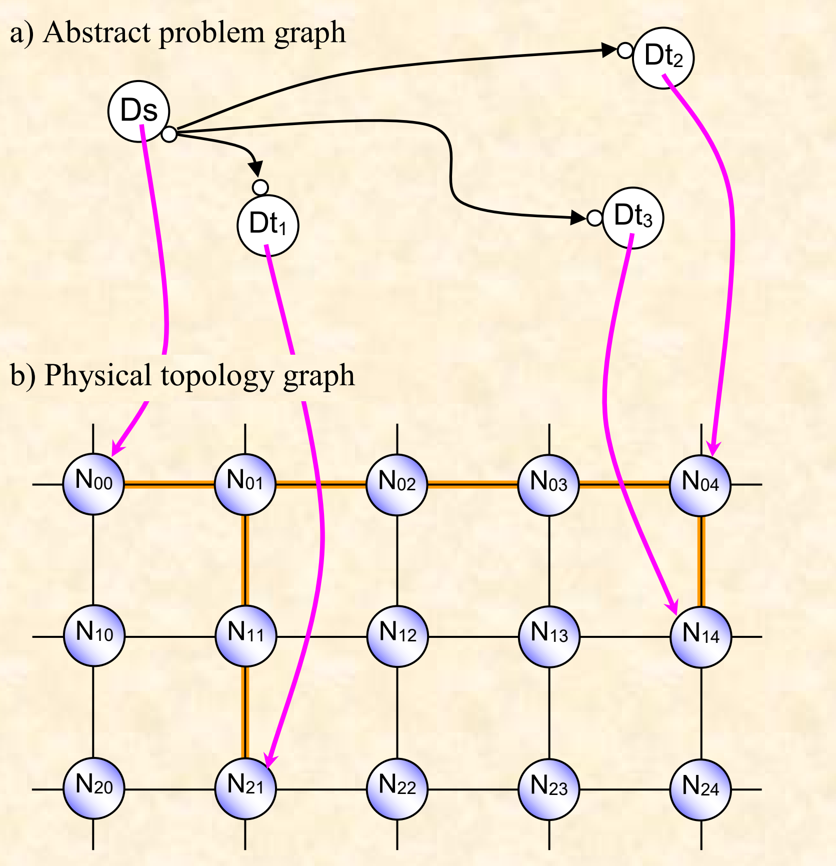
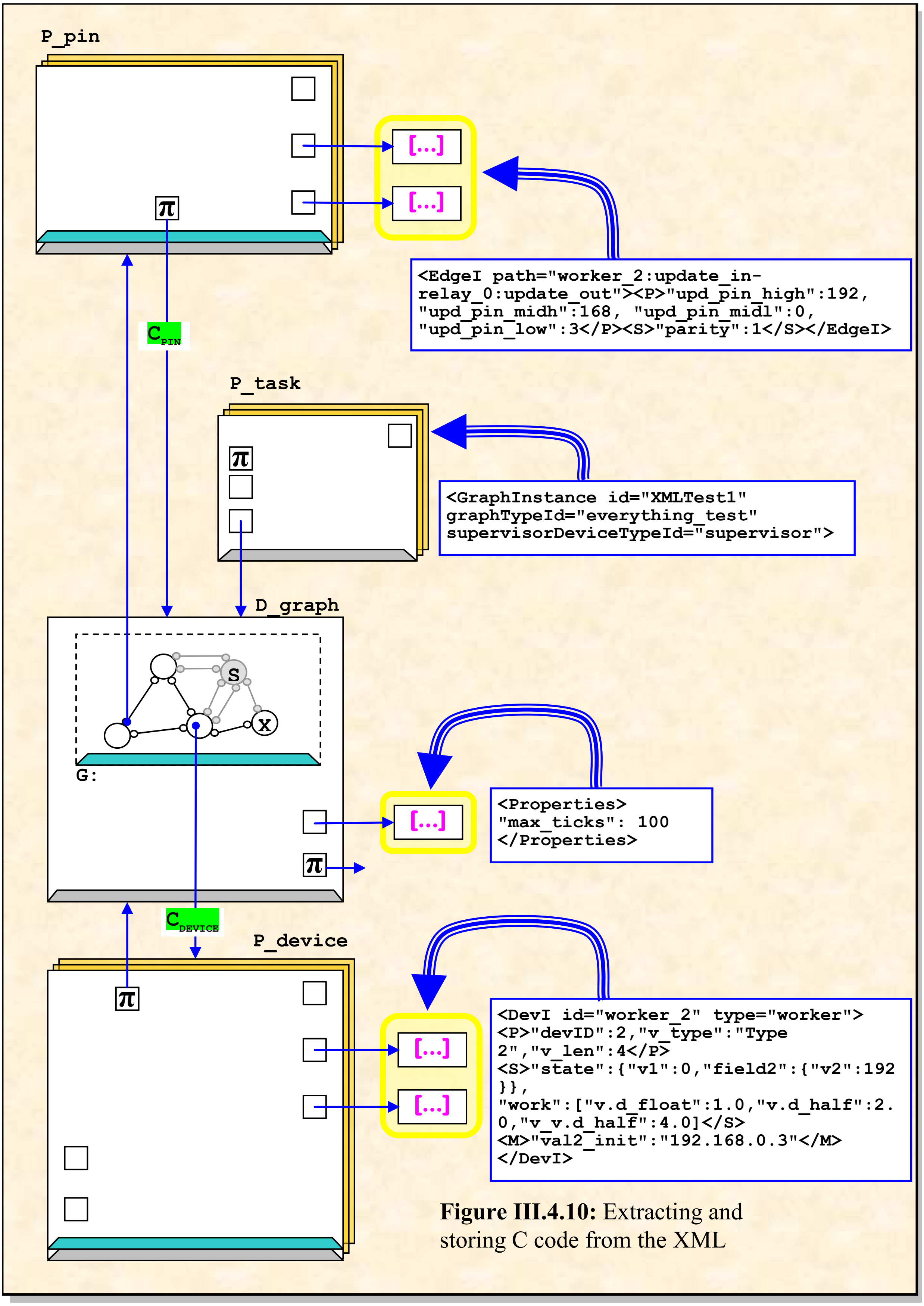


Figure III.13.1.a: Mapping application graph to topology graph



```
<EdgeI path="worker_2:update_in-
relay_0:update_out"><P>"upd_pin_high":192,
"upd_pin_midh":168, "upd_pin_midl":0,
"upd_pin_low":3</P><S>"parity":1</S></EdgeI>
```

```
<GraphInstance id="XMLTest1"
graphTypeId="everything_test"
supervisorDeviceTypeId="supervisor">
```

```
<Properties>
"max_ticks": 100
</Properties>
```

```
<DevI id="worker_2" type="worker">
<P>"devID":2,"v_type":"Type
2","v_len":4</P>
<S>"state":{"v1":0,"field2":{"v2":192
}},
"work":["v.d_float":1.0,"v.d_half":2.
0,"v_v.d_half":4.0]</S>
<M>"val2_init":"192.168.0.3"</M>
</DevI>
```

Figure III.4.10: Extracting and storing C code from the XML