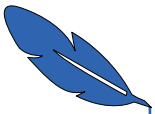


POETS

Orchestrator Internals

POETS Orchestrator architecture

Design notes

Volume III(B) – On Softswitches, the Supervisor and Composer: Bridging the gap to the Orchestrator

Ver 01	*24 October 2019*
Ver 02	*20 October 2020*
Ver 03	*20 February 2021*
Ver 04	*21 April 2021*

Graeme Bragg, Mark Vousden, Andrew Brown

Revision history

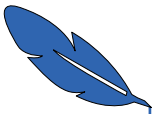
Revision	Date	Description
Ver 02	20 October 2020	Added Supervisor and Composer initial documentation.
Ver 03	20 February 2021	Added build control features, made sure that docs and source align, added some polish and beautification.
Ver 03	21 April 2021	Add documentation regarding the Supervisor's binary blob. Addressed comments from review.

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

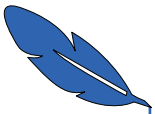
Throughout the lifetime of the project, the intention is that more and more details will be included, things will change, and the miracles will morph into hard quantitative details.

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.



Contents



1. Introduction

<Insert punchy intro here>

1.1. Related Documentation

This document discusses or refers to concepts that are more fully described in other documentation. Where this happens, the text will point you in the right direction. Other documents that may need to be consulted are:

- The Mothership documentation
- The Packet format documentation
- Orchestrator Documentation Volume III
- Tinsel documentation (https://github.com/POETSII/tinsel)

1.2. Document Structure

The remainder of this document is divided into five main sections: Section 2 details the inner-workings of the Softswitch; Section 3 provides an overview of the Supervisor, though this should be read in conjunction with the Mothership and Supervisor API documentation; Section 4 discusses the stand-alone compilation process for the Softswitches and Supervisor shared object involved in an application and details the required prerequisites; Section 5 describes the Composer, the Orchestrator component responsible for generating source files and compiling them; finally Section 6 discusses areas of future work and improvement.

2. The Softswitch

The Softswitch is the code that executes when a POETS application is run. It is a combination of general control code and code assembled from the device handler C-fragments and application definition provided by the XML. The Softswitch can be thought of as a very simple operating system that routes inbound packets to the correct device, sends any queued packets and executes device-specific handlers at appropriate times as shown in Figure 1. On a Tinsel-based POETS system, this is the code that executes on the RISC-V soft-cores with each hardware thread executing its own instance of the Softswitch.

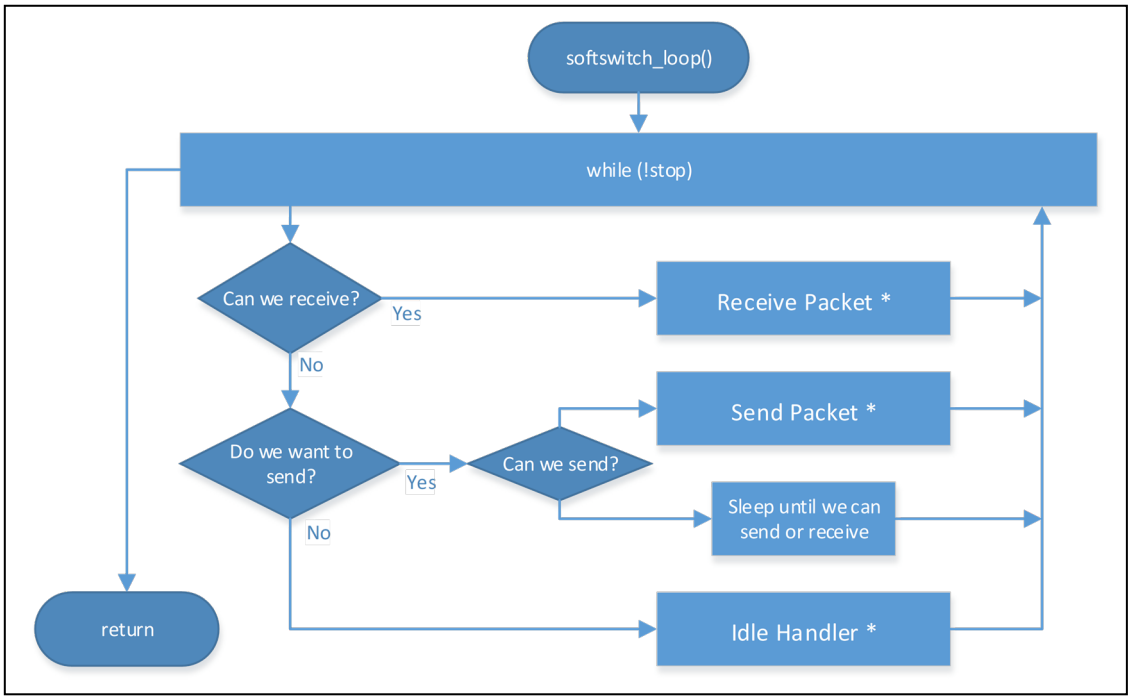


Figure 1: Abstract softswitch loop. Activities marked with a * call user-supplied handler code and execute the ReadyToSend handler.

The original Softswitch design was intended to host up to 1024 devices per Softswitch, however the data structure can support significantly more but actual real-world experience with Tinsel has shown that we may want to run considerably fewer devices per thread. The data structure supports hosting multiple device types on a single Softswitch; however the current implementation only supports executing a single device type per Tinsel core-pair[1] due to the limited instruction space available.

A note on terminology: In POETS Orchestrator parlance, the Softswitch deals in packets and not messages. Tinsel uses the term “message” to refer to what we would call packets. This document, the Softswitch source, Supervisor source and Composer uses the Orchestrator nomenclature where possible, however there are some instances where “message” is used to refer to a POETS Packet rather than a PMsg_p

[1] On Tinsel, instruction space is shared between pairs of cores. e.g. core 0 and core 1 share instruction space.

2.1. Operation

Once running, the Softswitch is fundamentally a large continuous control loop that that handles device interaction with the underlying hardware. This subsection describes the intricacies of this behaviour.

2.1.1. Operating Modes

The Softswitch can be operated in one of two modes (set at compile time) that affect how sent packets are handled:

In "Normal", or non-buffering, mode, a pin may only appear once in the list of pins wishing to send and the Softswitch assembles the packet to be sent (via a call to the pin's OnSend handler) immediately before it is emitted over the network. This means that the device's latest state used to assemble a packet and this may be different to the state that was used to determine that a pin wished to send a packet – if an application requires one packet per state update, then the application designer must account for this. A pin may only appear once in the list of pins wishing to send – it is up to the application to handle any required repeat transmissions. On Tinsel, the packet is assembled directly in the mailbox's send slot.

In Buffering mode, the Softswitch assembles the packet immediately after the application has determined that it wishes to send a packet. The assembled packet is added to a circular packet buffer and sent at a later time. A pin may appear multiple times in the ready-to-send list and packet buffer – unless the circular buffer is full, any call to the application's ReadyToSend handler that flags a pin as wanting to send will result in a packet being emitted at some point – a congested network may result in stale packets being emitted and applications should be designed to account for this.

2.1.2. Boot Sequence

Once the cores have been loaded by the Orchestrator (`init /app`), each thread begins to execute the Softswitch code. `softswitch_main()` begins the initialisation process by obtaining a pointer to the thread's `ThreadContext` structure and creating the Ready-to-Send circular buffer (`rtsBuf`). `softswitch_init()` is then called to continue initialisation.

`softswitch_init()` zeros all instrumentation and calls `device_init()` for each device hosted by the Softswitch. `device_init()` walks the device's data structure populating back pointers that are cannot be provided during code generation and then calls the device's `OnInit` handler.

Once initialisation is complete, the Softswitch enters `softswitch_barrier()`, which sends a packet to the Mothership to indicate that the thread has successfully booted. The thread will then block until it has received a packet with the `P_CNC_BARRIER` opcode set. This packet is sent by the Mothership after a `run /app` command is issued. When the Softswitch has been released from the barrier, the Softswitch enters `softswitch_loop()` after a short delay to allow other softswitches to start. The Softswitch remains in this loop until a packet with the `P_CNC_STOP` opcode is received, which causes the Softswitch to gracefully exit and stop execution.

2.1.3. Main Loop

Once entered, `softswitch_loop()` does not exit until `ThreadContext->ctlEnd` is non-zero. An overview of the default control flow in the non-buffering mode is shown in Figure 2. Detailed flow diagrams of the non-buffering and buffering modes are shown in Figure 3 and Figure 4 respectively. The order of some operations can be controlled at build time as detailed in Section 2.1.6.2.

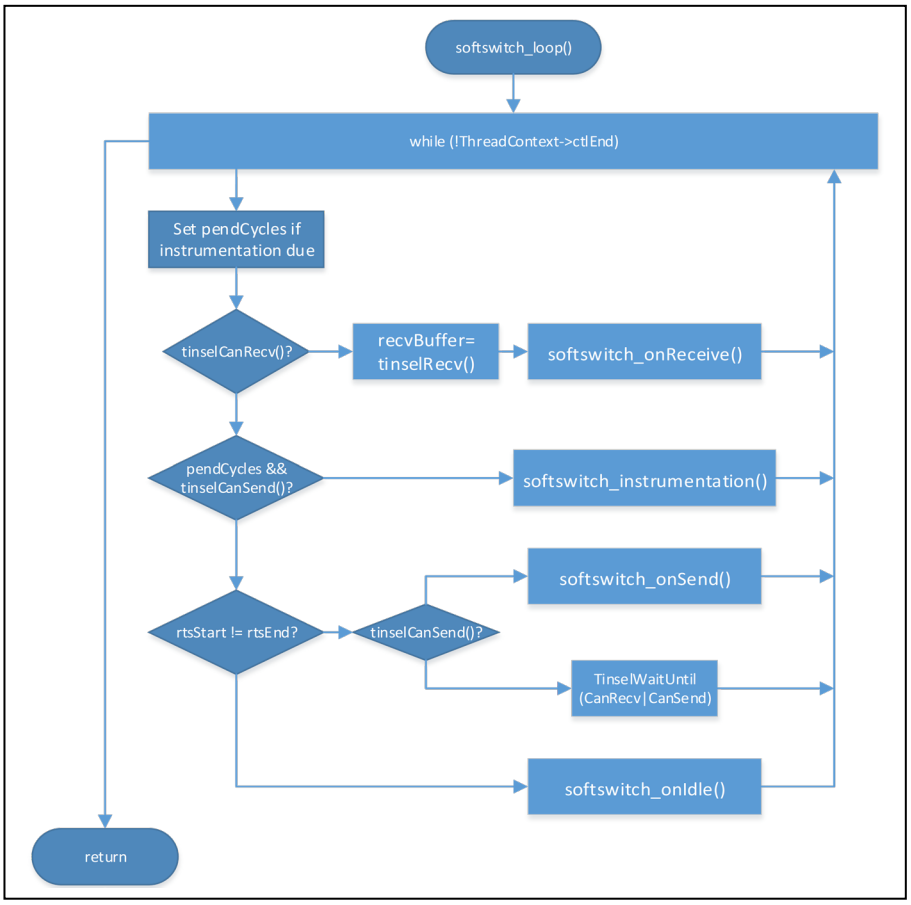


Figure 2: Abstract softswitch loop control flow: default order

2.1.4. Receive Handling

If there is a pending packet, the Softswitch retrieves it from the mailbox and calls `softswitch_onReceive()`. `softswitch_onReceive()` decodes the headers, handling internal control packets from the Mothership (e.g. stop or instrumentation request) directly without invoking any device-specific code, and handles the packet appropriately by calling the correct device receive or control handler.

The current version of the Softswitch supports unicast and broadcast packets. Unicast packets trigger the `OnReceive` handler for a single pin, or the `OnCtl` handler, on a single device. A broadcast packet triggers the `OnReceive` handler for a single pin, or the `OnCtl` handler, on all devices hosted on the Softswitch.

For unicast packets, the value of the device address is checked to ensure that it is within the range of devices hosted by the Softswitch. An out of range device address results in the packet being dropped and the generation of a log packet.

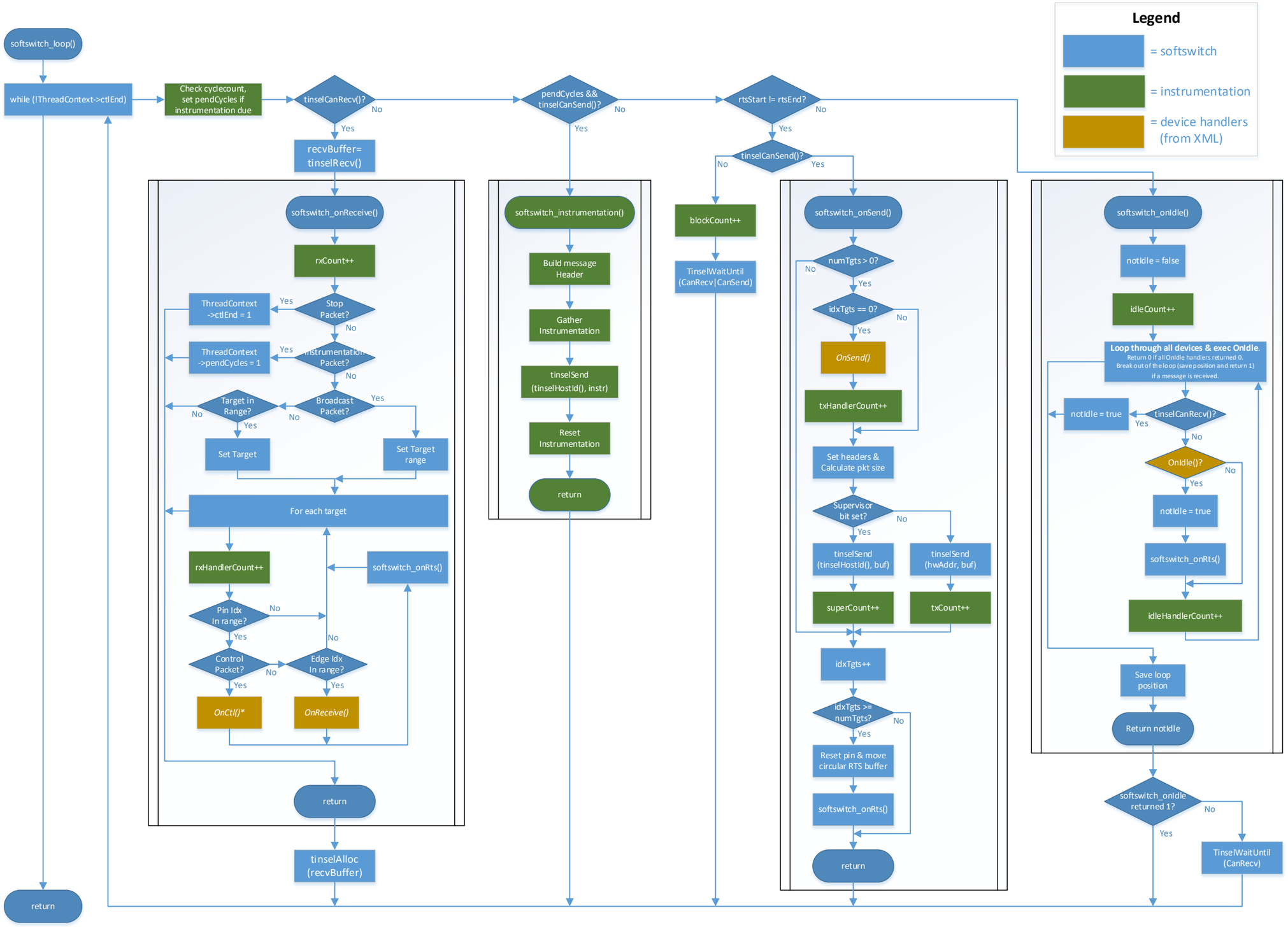


Figure 3: Detailed Softswitch loop control flow: Default execution order. See Figure 5 for `softswitch_onRts()` expansion.

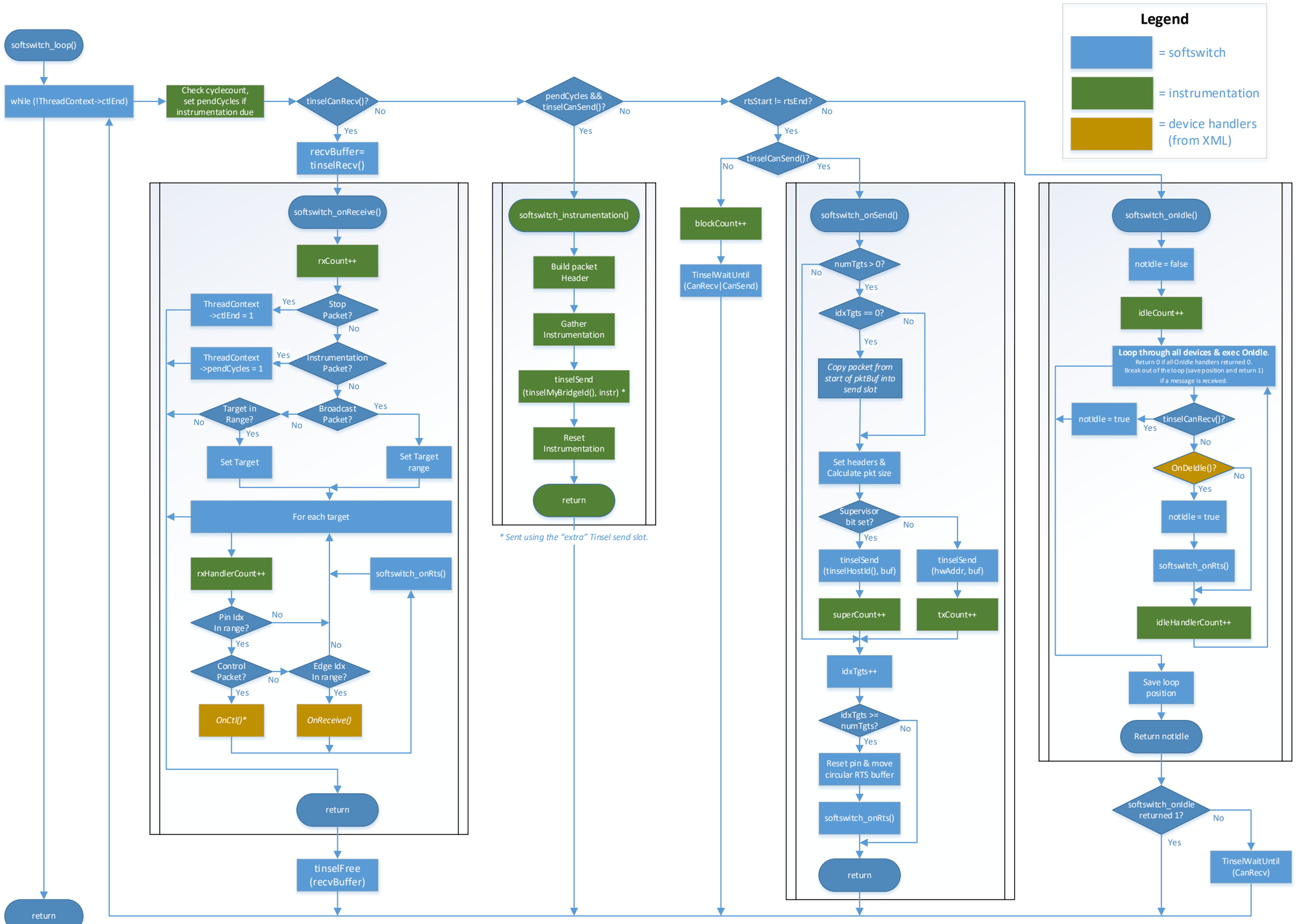


Figure 4: Detailed Softswitch loop control flow: Buffering mode. See Figure 5 for `softswitch_onRts()` expansion. `OnSend` occurs during `softswitch_onRts()`.

2.1.5. Ready to Send handling

`softswitch_onRTS()` is intended to be executed any time there may have been a change in a device's state (e.g. any time a packet is received, a pin has finished sending or OnInit/OnIdle/OnCompute returns a non-zero value). This uses the device's `ReadyToSend()` handler to generate a 32-bit-wide bit field that indicates which, if any, pins wish to send given the current device state. In non-buffering mode, the Softswitch adds any flagged pins, which do not already have a pending send, to the RTS list. In buffering mode, the Softswitch calls the `OnSend()` handler for any flagged pins, adding the generated packet payload(s) to the packet buffer if there is space. A detailed flow diagram of `softswitch_onRTS()` for both modes is shown in Figure 5.

The Softswitch uses an array of pointers-to-output-pins to keep track of which pins wish to send. The size of the array is set using `ThreadContext->rtsBufSize`[2]. The array is used as a circular buffer with ThreadContext->rtsStart holding the index of the first pending pin and `ThreadContext->rtsEnd` holding the index of the last pending pin. In buffering mode, a second array of 56-byte packets the same size as the RTS buffer exists and is used in parallel with the RTS buffer to track queued packets.

The following key points should be noted about the Ready to Send handling:

- A device can have no more than 32 output pins, including the implicit supervisor output pin if it has one.
- In non-buffering mode, **an individual pin may only appear once in the RTS list** and they appear in the order in which they were added to the list. The Softswitch has no concept of priority.
- In non-buffering mode, subsequent calls to `ReadyToSend()`that return a flag for a pin that already has a pending send (indicated by the pin's `sendPending` field) are ignored.
- In buffering mode, subsequent calls to `ReadyToSend()` when the circular buffer is full are ignored.
- It is not possible to remove a pin from the RTS list or a packet from the packet buffer once it has been added and a packet will, at some point, be sent.
- As per the GraphSchema specification, device state must not be updated within `ReadyToSend()`: the Softswitch prevents this by const-qualifying the state in the assembled ready to send handler.
- A pin that does not have any targets will not be added to the RTS list.

2.1.6. Send Handling

When the Softswitch determines that there is something to send (i.e. the RTS buffer start and end indices are not equal), the Softswitch checks to see if it is possible to send a packet. If it is not possible to send a packet, the Softswitch blocks on a call to `TinselWaitUntil()` until it is possible to either send or receive before returning to the start of the control loop. It should be noted that this has the potential to cause the Softswitch to deadlock during network congestion as `TinselWaitUntil()` does not (currently) support a timeout. However, if the Softswitch cannot send and cannot receive, it can do no useful work. In contrast, yielding to other threads may enable other threads to drain the network and clear the congestion. If it is possible to send a packet, `softswitch_onSend()` is called.

---

[2] See Section 5.5.2.7 for details as to how this is set. By default, for non-buffering mode this is calculated by Composer as the number of connected output pins hosted on the Softswitch, allowing all output pins to have a send pending, up to MAX_RTSBUFFSIZE. **This buffer must have a size greater than one** – by default, Composer will not create a buffer with fewer than MIN_RTSBUFFSIZE (10) slots. In buffering mode, the size of the buffer is set to MAX_RTSBUFFSIZE.

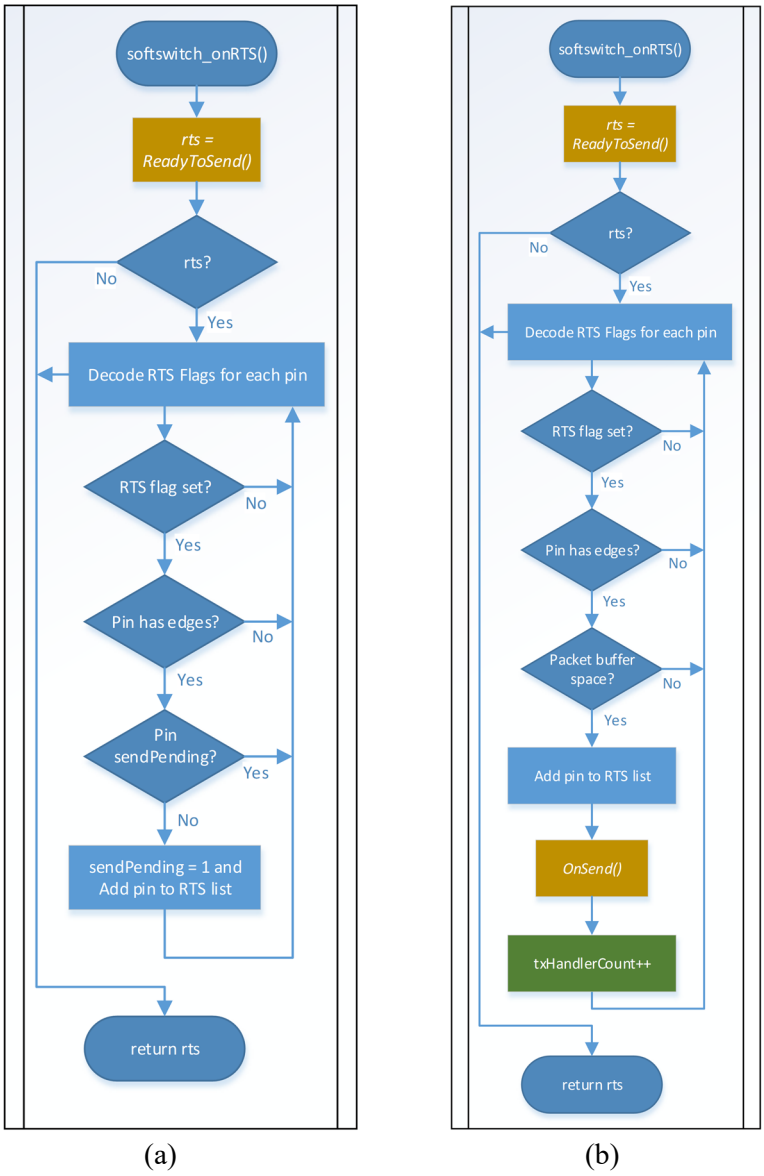


Figure 5: softswitch_onRTS execution flow for the normal mode of operation (a) and the buffering mode of operation (b)

Each time `softswitch_onSend()` is called, a single POETS packet is sent[3]. If a pin sends multiple packets (e.g. a pin has multiple targets), multiple calls to `softswitch_onSend()` are required. This prioritises receiving a packet over sending so that the network is drained as quickly as possible. The pin is only removed from the RTS list when the packet to the final target has been sent.

The first time `softswitch_onSend()` is called for a pending send for a pin, the payload for a packet is loaded into the last 56 bytes of the Tinsel send slot. The contents of a slot are persistent until re-written so this avoids a `memcpy` for each packet but nothing else can use the same slot – instrumentation and log packets make use of the "extra" send slot provided by Tinsel. In non-buffering mode, the slot is loaded by a call to the pin's `OnSend()` handler. In buffering mode, the slot

[3] Unless the target pin has no targets specified, in which case no packets are sent and the pin is removed from the RTS list.

is loaded by `memcpy`ing the payload from the packet buffer. Each call to `softswitch_onSend()` writes the header section of the packet to the slot.

2.1.6.1. Idle handling

When there is nothing to receive and nothing to send, the Softswitch will execute the `OnDeviceIdle` handler for each device hosted on the Softswitch. The Softswitch will step through each device it has executed the handler for all devices in one sitting or it is interrupted by a received packet.

A non-zero return from an `OnDeviceIdle` handler indicates that something interesting happened and `softswitch_onRTS()` is called for the device. The boilerplate code for `OnDeviceIdle` defaults to returning a non-zero value.

If the Softswitch completes an uninterrupted iteration over all devices and all of the `OnDeviceIdle` handlers returned zero, the Softswitch will block on a call to `TinselWaitUntil()` until it is possible to receive a packet. This stops the Softswitch from emitting instrumentation packets but yields processing time to other threads that may be active.

2.1.6.2. Loop Order

The ordering of the receive and instrumentation processing steps in the main loop can be controlled. By default, the Softswitch prioritises receive to constantly drain the network; however, this can result in no instrumentation being sent by devices that are inundated with inbound packets. By defining `SOFTSWITCH_PRIORITISE_INSTRUMENTATION` at build time, the Softswitch instead prioritises sending instrumentation packets if it is possible to do so over receiving packets.

2.1.7. Shutdown

When the Softswitch receives a packet with the `P_CNC_STOP` opcode, it exits the main loop at the earliest opportunity and gracefully shuts down with `softswitch_finalise()`. `softswitch_finalise()` clears the RTS list and returns control of all of the tinsel slots to the hardware. No confirmation packet is sent.

2.2. XML-derived Handlers

The Softswitch is an assembly of boilerplate code around user-supplied handler code from the XML. This section details how user-supplied handlers are used and which variables they have access to.

2.2.1. Reserved Variable Names

In addition to the variables listed in Section 2.2.2 the Softswitch makes use of several additional global variables and pre-processor macros. To avoid conflicts with the inner workings of the Softswitch, user-supplied handlers and code must not define or use any variable names or defines that begin with:

- P_
- p_
- __
- softswitch_

Additionally, user-supplied handlers and code must not define or set:

- LOG_BOARDS_PER_BOX
- LOG_CORES_PER_BOARD
- LOG_THREADS_PER_CORE

2.2.2. Available variables

Each handler has access to a set of pre-defined variables that can be referenced within the body of the handler.

Table I: Variables accessible within XML-derived handlers.

Variable Name	XML Handler(s)	Description
`graphProperties`	All	A pointer to the const-protected graphProperties structure. The layout is derived from the `GraphType`'s `<Properties>` section. Only available if defined in the XML.
`deviceProperties`	All	A pointer to the const-protected deviceProperties structure. The layout is derived from the `DeviceType`'s `<Properties>` section. Only available if defined in the XML.
`deviceState`	All	A pointer to the `deviceState` structure. The layout is derived from the `DeviceType`'s `<State>` section. This is const-protected in `ReadyToSend`. Only available if defined in the XML.
`edgeProperties`	`OnReceive`	A pointer to the const-protected `edgeProperties` structure. The layout is derived from the `InputPin`'s `<Properties>` section. Only available if defined in the XML.
`edgeState`	`OnReceive`	A pointer to the `edgeState` structure. The layout is derived from the `InputPin`'s `<State>` section. Only available if defined in the XML.
`message`	`OnReceive` `OnSend`	A pointer to the payload section of a packet, cast to the relevant format for the handler as defined in the XML.
`readyToSend`	`ReadyToSend`	A pointer to a 32-bit bit field that is used to determine which pins wish to send.
`ThreadContext`	All	A pointer to the `ThreadContext` structure. This has the layout detailed in Section 2.6. While available in handlers, it is not intended that XML-derived code will access this directly.
`deviceInstance`	All	A pointer to the `deviceInstance` structure for the current device. This has the layout detailed in Section 2.6. While available in handlers, it is not intended that XML-derived code will access this directly.
`pkt`	`OnReceive` `OnSend`	A `void*` pointer to the payload section of a `P_Pkt_t` stored in a Tinsel slot. While available in handlers, it is not intended that XML-derived code will access this directly.

2.2.3. Macros of Convenience

The Softswitch header provides several "Macros of convenience" to make writing handlers slightly easier. These macros are detailed in Table II.

Table II: Convenience macros that may be used within user-supplied handler code.

Macro	Evaluated to	Intended handler	Usage
`GRAPHPROPERTIES(x)`	`graphProperties->x`	All	Access to the GraphProperties struct.
`DEVICEPROPERTIES(x)`	`deviceProperties->x`	All	Access to the DeviceProperties struct for the current device.
`DEVICESTATE(x)`	`deviceState->x`	All	Access to the DeviceState struct for the current device.
`EDGEPROPERTIES(x)`	`edgeProperties->x`	OnReceive	Access to the EdgeProperties struct for the current edge.
`EDGESTATE(x)`	`edgeState->x`	OnReceive	Access to the EdgeState struct for the current pin.
`MSG(x)`	`message->x`	OnReceive OnSend	Access to the formatted packet payload for the input or output packet.
`PKT(x)`	`message->x`	OnReceive OnSend	Alternative to MSG()
`RTS(x)`	`*readyToSend |= RTS_FLAG_##x`	OnSend	Sets the RTS flag for the specified pin name. The name passed to this macro must match the name of the pin specified in the XML.
`RTSSUP()`	`*readyToSend |= RTS_SUPER_ IMPLICIT_SEND_FLAG`	OnSend	Sets the RTS flag for the implicit supervisor pin.

2.2.4. SharedCode

The Softswitch has access to the Graph Type's SharedCode section from the XML. If it exists in the XML, it is placed at the top of the generated source file immediately below the properties and state declarations. Shared code is intended to contain free functions, constant variables and defines that are shared across multiple handlers. This is an ideal place to place `#includes` and `#defines` that are common throughout the entire application.

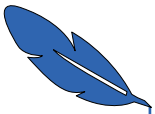
The user-supplied shared code exists at the top of the generated per-core source – it is not instanced per device. Shared code does not have access to the properties and state structures – any non-constant values that the functions require must be passed in as arguments.

2.2.5. ReadyToSend

A device's ReadyToSend handler is called any time something of interest occurs as a result of an XML-derived handler. This includes:

- OnInit returning "1".
- OnIdle/OnDeIdle returning "1".
- When an output pin has sent to its last target.
- After processing a received packet.

ReadyToSend has const-protected (i.e. read-only) access to the device's properties and state along with the graph's properties. While ReadyToSend has access to any variables or methods declared in the application's `<SharedCode>` section, it must not change any state. ReadyToSend should only update the bit field pointed to by `readyToSend` and any variables declared within ReadyToSend.



The bit field is defined as a uint32_t with each bit corresponding to an output pin. This means that there can be a total of 32 possible output pins (including the implicit supervisor output pin, if defined) per device type.

2.2.5.1. ReadyToSend Flags

A ReadyToSend bit flag variable (of the format `RTS_FLAG_<PINNAME>`) is automatically generated for each output pin defined in the XML. Additionally the `RTS_SUPER_IMPLICIT_SEND_FLAG` variable is available if the device has an implicit Supervisor output pin.

The generated ReadyToSend flags should be ORed with the bit field pointed to by `readyToSend` to indicate that a pin wishes to send. E.g. `*readyToSend |= RTS_SUPER_IMPLICIT_SEND_FLAG`

2.2.6. OnInit

A device's OnInit handler is called during initialisation before the Softswitch reaches the barrier. OnInit can be used to carry out any initialisation that cannot be handled with the generated properties and state initialisers.

Returning 1 triggers a call to ReadyToSend. By default, the handler returns 1 after the XML-derived C fragment has been executed.

2.2.7. OnReceive

Each pin type has an associated OnReceive handler that is called when a packet is received for that pin. In addition to the device properties, device state and graph properties, the handler has access to the specific edge state and edge properties for the specific edge that triggered the handler. The payload of the packet is accessed through the `message` variable (or by using the `MSG()` or `PKT()` macros).

2.2.8. OnSend

OnSend prepares a packet to be sent with user-supplied code populating the payload of the packet. In the non-buffering more, OnSend is called immediately before the first packet for a pin is sent and forms the packet directly in the Tinsel send slot. In buffering mode, the packet is formed in a packet buffer immediately after the ReadyToSend handler completes. The payload of the packet is accessed through the `message` variable (or by using the `MSG()` or `PKT()` macros).

2.2.9. OnDeviceIdle

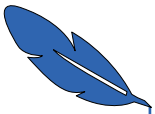
A device's OnIdle handler may be called any time the Softswitch has no packets to send or receive. There is no guarantee on when OnIdle is called or how many times it will be called, only that it will be called eventually.

2.2.10. OnHWIdle

Called when the hardware idle detection triggers. Not currently implemented.

2.2.11. OnCtl

Called when a packet with a user-defined opcode is received. Not currently implemented.



2.3. Device Identification

Devices have two indices that uniquely identify them in different contexts. The first of these is the thread-unique Device ID. This is the value used in the Device Address section of the `swAddr` header and uniquely identifies the device in the application when combined with the hardware address.

The second of these is the Supervisor-unique Device Index. The `deviceIdx` uniquely identifies the device within the scope of a Supervisor (i.e. within a box). The `deviceIdx` is used to populate the `pinAddr` header for control packets where the destination needs to know which device sent a packet. This includes log packets and packets sent to the Supervisor's implicit receive pin.

2.4. Log Packets

`handler_log()` is a method provided by the Softswitch to allow user-supplied handler code to send log messages to the Logserver via the Mothership. These packets have the `P_CNC_LOG` Opcode set. Log messages may be up to four packets long, with the length of message that can be encapsulated dependent on the specific log handler that the Softswitch has been compiled with.

`handler_log()` is declared as a variadic macro in `softswitch.h` with the following expected call pattern:

```
void handler_log(int level, const char * pkt, F... args)
```

`handler_log()` calls `__handler_log()`, a variadic template method with the following signature:

```
inline void handler_log(uint32_t src, int level, const char * pkt, F... args)
```

where:

`level` indicates the priority of the message with higher values indicating higher priority messages. The Softswitch can be configured at compile time to ignore messages with priorities below a given threshold by defining `P_LOG_LEVEL`. By default, packets with priority 2 or greater are sent by the Softswitch.

`pkt` is a pointer to the c-string representation of the log message. In `printf` parlance, this is the format string.

`args` are the values that the log message is meant to encode.

`src` is taken from the device's `devInst_t->deviceIdx` field and uniquely identifies the device on the supervisor

The first byte of the payload of the emitted packet is used to send a decrementing sequence number to facilitate fragmented log packets. E.g. the first log packet has the greatest sequence number and the last packet always has a sequence number of zero. A struct for the format of the log packet payload is provided in `poets_pkt.h`.

The Softswitch currently includes one log handler (the trivial log handler), which can be enabled at compile time by defining `TRIVIAL_LOG_HANDLER` at compile time.

2.4.1. Trivial log Handler

The trivial log handler only sends the format string portion of the provided log packet – all additional arguments and values are ignored. This gives a per-packet payload of 55 characters and a total maximum packet length of 220 characters, including the trailing null character. Log messages longer than 219 characters are truncated to 219 characters and sent.

2.5. Instrumentation

The Softswitch includes basic thread-level instrumentation that can be enabled by defining `SOFTSWITCH_INSTRUMENTATION` at build-time. Additional per-core instrumentation counters (e.g. cache hit/miss/writeback counts and CPU Idle count) are enabled if `TinselEnablePerfCount == true`. Each instrumentation packet includes the sending thread's cache of the per-core counters.

The current set of instrumentation is designed to fit in a single packet that is sent roughly every `P_INSTR_INTERVAL` cycles (by default this is defined as 240000000 cycles, giving an instrumentation packet roughly every second on Tinsel 0.6.3).

A struct for the format of an instrumentation packet payload is provided in `poets_pkt.h` and is summarised in Table III.

Table III: Instrumentation packet layout, including header field specification.
32-bits of payload remain unused for future use.

Field	Type	Description
`swAddr`	`uint32_t`	Software address. Mothership and CNC bits set. Opcode set to P_CNC_INSTR.
`pinAddr`	`uint32_t`	Set to the Hardware address of the sending Softswitch.
`cIDX`	`uint32_t`	Index (sequence) of the instrumentation packet.
`cycles`	`uint32_t`	Cycle count since last instrumentation packet.
`rxCnt`	`uint32_t`	Number of POETS packets received.
`rxHanCnt`	`uint32_t`	Number of times a device OnReceive handler has been called.
`txCnt`	`uint32_t`	Number of POETS packets sent to other devices.
`supCnt`	`uint32_t`	Number of POETS packets sent to motherships.
`txHanCnt`	`uint32_t`	Number of times a device OnSend handler has been called.
`idleCnt`	`uint32_t`	Number of times Softswitch_onIdle called.
`idleHanCnt`	`uint32_t`	Number of times a device OnCompute handler has been called.
`blockCnt`	`uint32_t`	Number of times the mailbox has indicated that it is not possible to send a packet.
`missCount`	`uint32_t`	Cache miss count[4].
`hitCount`	`uint32_t`	Cache hit count[4].
`writebackCount`	`uint32_t`	Cache writeback count[4].
`CPUIdleCount`	`uint32_t`	CPU Idle count[4].

2.6. Data Structure

The Softswitch's data structure is split between the top and bottom of the thread's memory partition. The ThreadContext is stored at the base of what Tinsel refers to as the heap and the rest of the data structure is stored on the stack. Figure 6 shows the data structure and its linkage.

The ThreadContext (Table IV) anchors the rest of the data structure and is stored at the base of the DRAM "heap". A pointer to the `ThreadContext` is obtained during Softswitch initialisation with `tinselHeapBase()`, which is then `static_cast`ed to a `PThreadContext*`.

Pointers to all instances of properties and state throughout the data structure are stored as void pointers as the data structure has no knowledge of (and need to know) the underlying structure of the application-specific properties and state. The generated code `static_cast`s the void pointers to the correct application-specific typing as required, e.g. within the boilerplate code that is pre-pended to device handlers.

[4] Since the last instrumentation packet was sent.

Table IV: Thread Context. Defined in softswitch_common.h as PThreadContext/ThreadCtxt_t.

Field	Type	Description
`numDevTyps`	`uint32_t`	Number of device types on this Softswitch[5].
`devTyps`	`devTyp_t*`	Pointer to an array of device types.
`numDevInsts`	`uint32_t`	Number of devices hosted by this Softswitch.
`devInsts`	`devInst_t*`	Pointer to an array of device instances.
`properties`	`const void*`	Graph global properties.
`rtsBufSize`	`uint32_t`	The size of the RTS circular buffer.
`rtsBuf`	`outPin_t**`	Pointer to the RTS circular buffer.
`pktBuf`	*`P_Pkt_pyld_t*`*	Pointer to the send packet buffer. *Only exists in buffering mode.*
`rtsStart`	`uint32_t`	Index of the first entry in the RTS buffer.
`rtsEnd`	`uint32_t`	Index of the last entry in the RTS buffer.
`idleStart`	`uint32_t`	Index of which device to start `softswitch_OnIdle` from.
`ctlEnd`	`uint32_t`	Loop control value: 0 = stop, 1 = run.
`lastCycles`	`uint32_t`	Cycle count last time instrumentation was sent.
`pendCycles`	`uint32_t`	Indicates whether it is time to send instrumentation.
`txCount`	`uint32_t`	Number of packets sent[6].
`superCount`	`uint32_t`	Number of Supervisor packets sent[6].
`rxCount`	`uint32_t`	Number of packets received over the network[6].
`txHandlerCount`	`uint32_t`	Number of `OnSend` handlers called[6].
`rxHandlerCount`	`uint32_t`	Number of `OnReceive` handlers called[6].
`idleCount`	`uint32_t`	Number of times `softswitch_OnIdle` called[6].
`idleHandlerCount`	`uint32_t`	Number of times `OnIdle`/`OnCompute` called[6].
`blockCount`	`uint32_t`	Number of times network has been blocked[6].
`cycleIdx`	`uint32_t`	Index of the instrumentation packet.
`lastmissCount`	`uint32_t`	Cache miss count[7].
`lasthitCount`	`uint32_t`	Cache hit count[7].
`lastwritebackCount`	`uint32_t`	Cache writeback count[7]
`lastCPUIdleCount`	`uint32_t`	CPU Idle count[7].

Initialisers for all thread-specific structures are placed in the generated per-thread code (vars_<core>_<thread>.cpp). The initialiser for the GraphProperties is placed in vars_<core>.cpp, which is common for all threads on a core.

The initialiser for the ThreadContext relies on an attribute to place it in the correct section of the compiled binary:

```
__attribute__((section (".thr<THREADNUMBER>_base")))
```

where `THREADNUMBER` is the number of the thread on the core, e.g. Thread 0 is `".thr0_base"`. This is a GCC-specific option, `#pragma section()` and `__declspec(allocate))` may provide equivalent functionality under MSVC.

Data that is common across all devices of the same type (e.g. handler pointers, size of properties/state, pin types, etc.) are stored in structures shown in the definitions section and detailed in Section 2.6.1. Data that is device-specific (e.g. pointers to device/edge properties/state, pin targets, etc.) are stored in the structures shown in the instance section and detailed in Section 0

[5] This will always be 1. A softswitch can only host a single device type.
[6] Reset each time an instrumentation packet is sent.
[7] Since the last time instrumentation was sent.

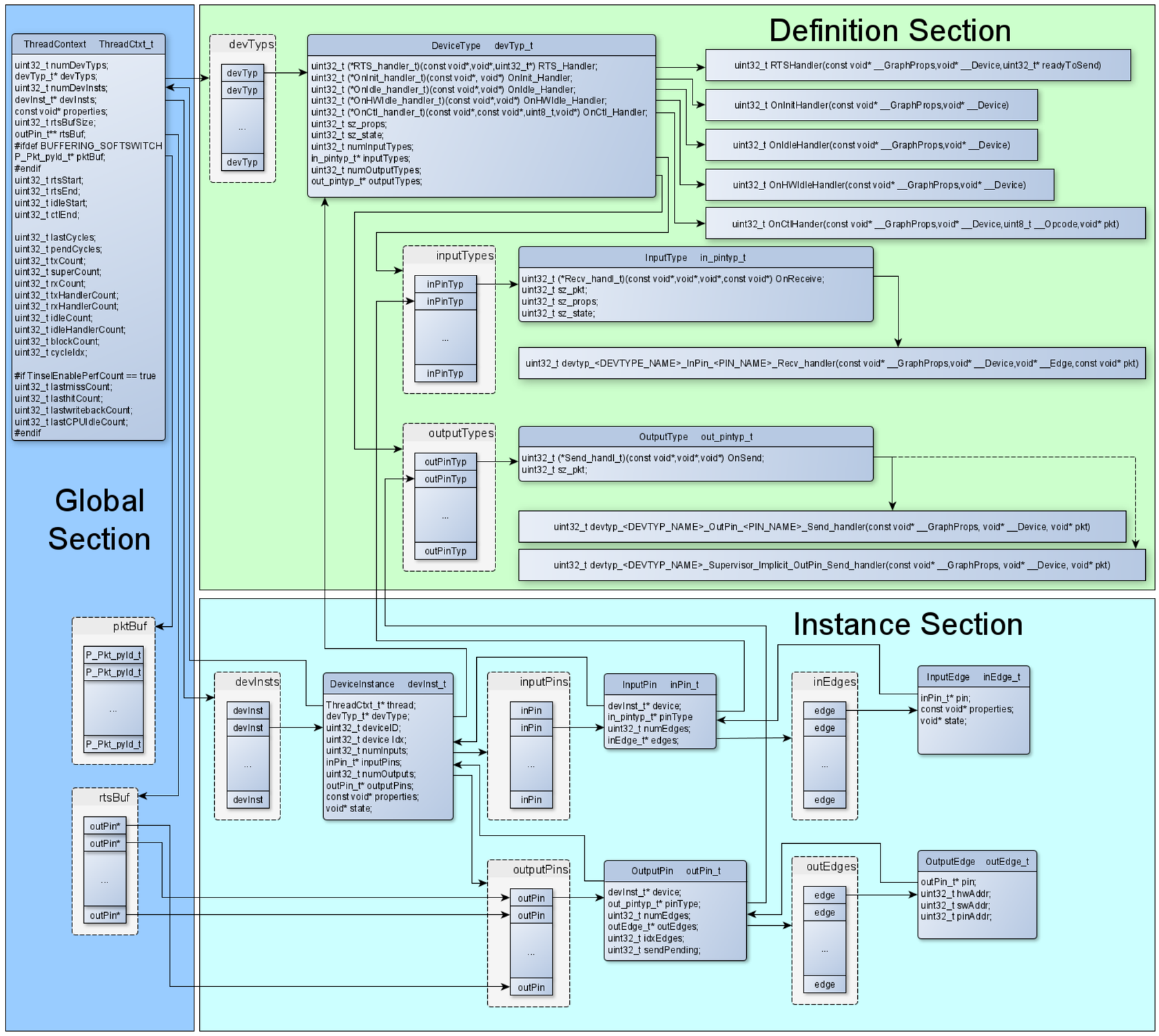


Figure 6: Softswitch Data structure. The `pktBuf` array only exists if the Softswitch has been compiled in buffering mode.

2.6.1. Definitions Section

The definitions section of the Softswitch data structure contains structures that define the behaviour of devices hosted on the Softswitch.

The definitions section of the data structure is anchored around an array of device type structs (Table V). Each device type struct contains pointers to the device-level handlers; the size of the properties and state structs; the number of input pins and output pins the device type has; and pointers to arrays of input pin and output pin types.

The input pin type struct (Table VI) contains a function pointer to the pin type's receive handler and the sizes of the packet, edge properties and edge state. The output pin type struct (Table VII) contains a pointer to the send handler and the size of the packet.

Table V: Device Type struct. Defined in softswitch_common.h as PDeviceType/devTyp_t

Field	Type	Description
`RTS_Handler`	`RTS_handler_t`	Function pointer to the device type's RTS handler.
`OnInit_Handler`	`OnInit_handler_t`	Function pointer to the device type's OnInit handler.
`OnIdle_Handler`	`OnIdle_handler_t`	Function pointer to the device type's OnIdle handler.
`OnHWIdle_Handler`	`OnHWIdle_handler_t`	Function pointer to the device type's OnHWIdle handler.[8]
`OnCtl_Handler`	`OnCtl_handler_t`	Function pointer to the device type's OnCtl handler.[8]
`sz_props`	`uint32_t`	Size in bytes of the device type's properties.
`sz_state`	`uint32_t`	Size in bytes of the device type's state.
`numInputTypes`	`uint32_t`	Number of input pin types the device type has.
`inputTypes`	`in_pintyp_t*`	Array of input pin types. Devices pins point to this.
`numOutputTypes`	`uint32_t`	Number of output pin types the device type has.
`outputTypes`	`out_pintyp_t*`	Array of output pin types. Devices pins point to this.

Table VI: Input `PinType` struct. Defined in `softswitch_common.h` as `PInputType/in_pintyp_t`

Field	Type	Description
`Recv_handler`	`Recv_handler_t`	Function pointer to the input's receive handler.
`sz_pkt`	`uint32_t`	Size in bytes of the packet.
`sz_props`	`uint32_t`	Size of the edge's properties.
`sz_state`	`uint32_t`	Size of the edge's state.

Table VII: Output `PinType` struct. Defined in `softswitch_common.h` as `POutputType/out_pintyp_t`

Field	Type	Description
`Send_Handler`	`Send_handler_t`	Function pointer to the input type's send handler.
`sz_pkt`	`uint32_t`	Size in bytes of the packet.

[8] Currently not fully implemented

2.6.2. Instance Section

The instance section contains all of the device-specific data for each device hosted on the Softswitch.

The instance section is anchored around an array of device instance structs (Table VIII). Each device instance includes a back pointer to the thread context, a pointer to the device type, the device's ID on this thread and the device's Supervisor-unique index. The struct also contains pointers to an array of output pin structs (Table IX), an array of input pin structs (Table XI), properties and state along with the sizes of the two arrays.

Table VIII: Device Instance struct. Defined in `softswitch_common.h` as `PDeviceInstance/devInst_t`

Field	Type	Description
`thread`	`PThreadContext*`	Back pointer to the ThreadContext.
`devType`	`const devTyp_t*`	Pointer to the Device Type.
`deviceID`	`uint32_t`	Thread-unique ID (index) of the device.
`deviceIdx`	`uint32_t`	Supervisor-specific device index.
`numInputs`	`uint32_t`	Number of inputs the device has.
`inputPins`	`inPin_t*`	Pointer to the inputPin array.
`numOutputs`	`uint32_t`	Number of outputs the device has.
`outputPins`	`outPin_t*`	Pointer to the outputPin array.
`properties`	`const void*`	Pointer to the device's properties.
`state`	`void*`	Pointer to the device's state.

Table IX: Input pin instance struct. Defined in `softswitch_common.h` as `PInputPin/inPin_t`

Field	Type	Description
`device`	`const PDeviceInstance*`	Back pointer to the device instance.
`pinType`	`in_pintyp_t*`	Pointer to the pin type.
`numEdges`	`uint32_t`	Number of sources.
`inEdges`	`inEdge_t*`	Pointer to array of input edge structs (Table X)

Table X: Input edge instance struct. Defined in `softswitch_common.h` as `PInputEdge/inEdge_t`

Field	Type	Description
`pin`	`const PInputPin*`	Back pointer to pin the pin instance.
`properties`	`const void*`	Pointer to the edge properties.
`state`	`void*`	Pointer to the edge state.

Table XI: Output pin instance struct. Defined in `softswitch_common.h` as `POutputPin/outPin_t`

Field	Type	Description
`device`	`PDeviceInstance*`	Back pointer to the device instance.
`pinType`	`out_pintyp_t*`	Pointer to the pin type.
`numEdges`	`uint32_t`	Number of targets the pin has.
`outEdges`	`outEdge_t*`	Pointer to the array of output edge structs (Table XII).
`idxEdges`	`uint32_t`	Index of the next edge to send on
`sendPending`	`uint32_t`	Flag indicating the pin wants to send.

Table XII: Output edge instance struct. Defined in `softswitch_common.h` as `POutputEdge/outEdge_t`

Field	Type	Description
`pin`	`POutputPin*`	Back pointer to pin the pin instance.
`hwAddr`	`uint32_t`	Target Hardware Address.
`swAddr`	`uint32_t`	Target Software Address.
`pinAddr`	`uint32_t`	Target Pin Address.

2.7. Configuration Options

The compilation of the Softswitch sources can be influenced by several pre-processor defines. For convenience, Table XIII summarises these options, their location, their default and where their use is discussed in this document.

Table XIII: Softswitch configuration parameters

Definition	Location	Default	Section Reference
Generation			
`MAX_RTSBUFFSIZE`	`Composer.cpp`	*4096*	Footnote 2
`MIN_RTSBUFFSIZE`	`Composer.cpp`	*10*	Footnote 2
Build			
`DISABLE_SOFTSWITCH_INSTRUMENTATION`	<environment/ Makefile>	*undefined*	2.5
`SOFTSWITCH_PRIORITISE_INSTRUMENTATION`	<environment/ Makefile>	-	2.1.6.2
`TRIVIAL_LOG_HANDLER`	<environment/ Makefile>	*defined*	2.4
`P_LOG_LEVEL`	`softswitch.h`	*2*	2.4
`TinselEnablePerfCount`	(tinsel) `config.h`	*true*	2.5

2.8. Alternative Backends

While designed to be used with Tinsel, it is possible to use the Softswitch with alternative backends. This requires minimal changes to the underlying code. This has proved to be very useful for debugging and application development with an MPI-based backend.

Any alternative backend must provide a `tinsel.h` that abstracts the tinsel methods listed in Table XIV and provides the defines/consts listed in Table XV for the Softswitch to operate without modification. Alternative backends that require initialisation and/or de-initialisation must also define the implementation-specific macros listed in Table XVI.

The backend must also provide the following enum and method:

```
typedef enum {TINSEL_CAN_SEND = 1, TINSEL_CAN_RECV = 2} TinselWakeupCond;

#ifdef __cplusplus
INLINE TinselWakeupCond operator|(TinselWakeupCond a, TinselWakeupCond b)
{
    return (TinselWakeupCond) (((uint32_t) a) | ((uint32_t) b));
}
#endif
```

Table XIV: Backend methods required by the Softswitch. Different methods are required for Tinsel 0.6 and 0.8.

Method	Tinsel Version	Description
`uint32_t tinselId()`		Return the Hardware Address of the current thread.
`uint32_t tinselHostId()`		Return the Hardware Address of the Mothership.
`uint32_t tinselMyBridgeId()`	0.8	Return the Hardware Address of the Mothership for this core.
`void* tinselHeapBase()`		Return pointer to base of thread's "heap" section.
`volatile void* tinselSlot(int n)`	0.6	Get pointer to message-aligned slot in "mailbox" scratchpad.
`volatile void* tinselSendSlot()`	0.8	Get pointer to the send slot.
`volatile void* tinselSendSlotExtra()`	0.8	Get pointer to the extra send slot.
`void tinselAlloc(volatile void* addr)`	0.6	Give the mailbox permission to use given address to store a message.
`void tinselFree(volatile void* addr)`	0.8	return a receive slot to the hardware.
`int tinselCanRecv()`		Determine if calling thread can receive a message.
`volatile void* tinselRecv()`		Receive message.
`int tinselCanSend()`		Determine if calling thread can send a message.
`void tinselSetLen(int n)`		Set the length (in flits) to be sent.
`void tinselSend(int dest, volatile void* packet)`		Send message at packet to dest. packet must be a pointer to a slot.
`uint32_t tinselUartTryPut(uint8_t x)`		Send byte to host (over DebugLink UART) (Returns non-zero on success).
`void tinselWaitUntil(TinselWakeupCond cond)`		Suspend thread until wakeup condition satisfied.
`uint32_t tinselCycleCount()`		Get the Cycle count performance counter.
`uint32_t tinselMissCount()`		Get the Cache miss performance counter.
`uint32_t tinselHitCount()`		Get the Cache hit performance counter.
`uint32_t tinselWritebackCount()`		Get the Cache writeback counter.
`uint32_t tinselCPUIdleCount()`		Get the CPU Idle performance counter.

Table XV: Defines/consts required by the Softswitch

Define	Type	Description
`TinselMaxFlitsPerMsg`	integer	The number of flits each packet is divided into.
`TinselLogMsgsPerThread`	integer	Base 2 log of the number of packet slots per thread.
`TinselLogBytesPerMsg`	integer	Base 2 log of the size of the packet's payload.
`TinselLogBytesPerFlit`	integer	Base 2 log of the size of each flit.
`TinselMeshXBitsWithinBox`	integer	Number of bit used for each box X address
`TinselMeshYBitsWithinBox`	integer	Number of bit used for each box Y address
`TinselMeshXLenWithinBox`	integer	Width of the board grid within a box.
`TinselMeshYLenWithinBox`	integer	Height of the board grid within a box.
`TinselLogCoresPerBoard`	integer	Base 2 log of the number of cores per box.
`TinselLogThreadsPerCore`	integer	Base 2 log of the number of hardware threads per core.
`TinselEnablePerfCount`	boolean	Indicates whether the Tinsel perf counters are enabled
`TinselClockFreq`	integer	Core clock frequency in MHz.

Table XVI: Initialisation and de-initialisation macros.

Macro	Description
`BACKEND_INIT(a)`	Backend initialisation. Called before entry into `softswitch_main()`.
`BACKEND_DEINIT()`	Backend teardown. Called after return from `softswitch_main()`.

3. The Supervisor

The Supervisor is a shared object that is loaded by the Mothership at application initialisation. It provides application-specific behaviour for a set of pre-defined methods that are called at specific points in the Mothership's execution. The Supervisor executes in x86-land with access to the underlying filesystem and network – this means that user-supplied code can read/write files, access network shares, etc.

The callable methods contained within the Supervisor shared object are summarised in Table XVII. All of the methods, with the exception of `SupervisorIdx2Addr()`, `GetSupervisorAddresses()` and `GetSupervisorApi()`, return non-zero if an error occurs.

Table XVII: Supervisor shared object methods

Method	Description
`int SupervisorInit()`	Called to initialise the Supervisor. Allocates the properties and state. Calls the Supervisor's OnInit handler.
`int SupervisorCall(std::vector<P_Pkt_t>&, std::vector<P_Addr_Pkt_t>&)`	Method for processing received packets. This will call the correct receive handler to process the inbound packet, either the implicit receive handler or a pin-specific handler (when implemented). SupervisorCall takes a reference to a vector of input packets and a reference to a vector of output addressed packets. The vector of addressed packets will contain entries if there are packets to be sent.
`int SupervisorIdle()`	A method that may be called periodically when the Mothership has no other work to do. Calls the Supervisor's `OnIdle` handler.
`int SupervisorCtl()`	A method to handle received messages. This encapsulates the Supervisor's `OnCTL` handler. [9]
`int SupervisorRTCL()`	A method to handle events emitted by the RTCL. Calls the Supervisor's `OnRTCL` handler. [9]
`int SupervisorExit()`	A method to gracefully tear down the Supervisor during Orchestrator exit. Calls the Supervisor's `OnStop` handler.
`uint64_t SupervisorIdx2Addr(uint32_t)`	A method to convert a device index to a full-symbolic address.
`void GetSupervisorAddresses (std::vector<SupervisorDeviceInstance_t>&)`	A method to provide a copy of the `DeviceVector` to the Mothership.
`SupervisorApi* GetSupervisorApi()`	A method to return a pointer to the Supervisor API contained within the Supervisor.

The front-end methods interact with a class (Supervisor, Figure 7) that encapsulates all of the application-specific behaviour. All of the class members are static and the definitions of all of the methods along with the `SupervisorProperties_t` and `SupervisorState_t` structs are contained within the generated source code. All of the methods must be present in the generated source file, even if they only contain a "`return 0;`" stub.

[9] Not currently implemented.

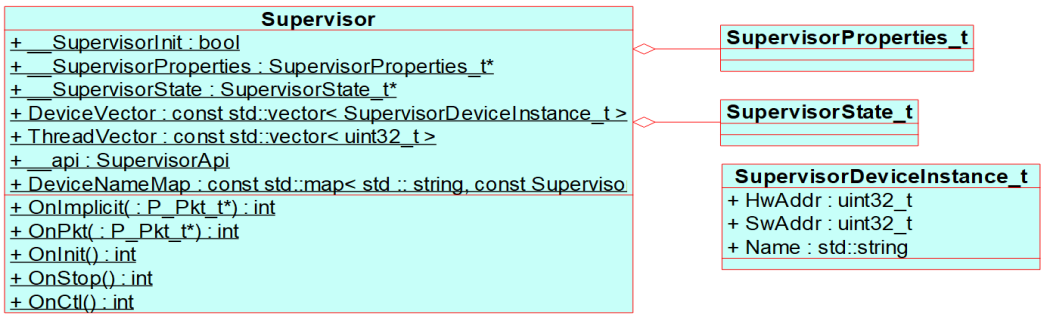


Figure 7: Supervisor class diagram.

3.1. Supervisor API

Application writers have access to a Supervisor API (encapsulated within the `Super` namespace) that provides several utility methods. The API is provisioned by the Mothership during initialisation. For further details, see the Supervisor API section of the Mothership documentation.

3.2. XML-derived Handlers

Like the Softswitch, the Supervisor is a combination of boilerplate code wrapped around user-supplied handlers derived from the application's XML definition. This subsection details the variables available within user-supplied code, reserved variable names and how the XML-derived code fragments are used.

3.2.1. Available variables

Each handler has access to a set of pre-defined variables (Table XVIII) that can be referenced within the body of the handler.

Table XVIII: Variables accessible within XML-derived handlers.

Variable Name	XML Handler(s)	Description
`graphProperties`	All	A pointer to the const-protected graphProperties structure. The layout is derived from the GraphType's `<Properties>` section.[10]
`supervisorProperties`	All	A pointer to the const-protected supervisorProperties structure. The layout is derived from the Supervisor's `<Properties>` section.[10]
`supervisorState`	All	A pointer to the supervisorState structure. The layout is derived from the Supervisor's `<State>` section. [10]

3.2.2. Reserved Variable Names

In addition to the variables listed in Table XVIII, the Supervisor makes use of several additional variables. To avoid conflicts with the inner workings of the Supervisor, user-supplied handlers and code must not define or use any variable names or defines that begin with:

- P_
- p_
- __

Additionally, user-supplied handlers and code must not define or set:

- DeviceVector
- CoreVector

[10] Only available if defined in the XML

3.2.3. Macros of Convenience

The Supervisor provides several pre-processor macros to provide more convenient access to the underlying Supervisor data structures for application developers. The macros exist in `supervisor_generated.h` and are created as part of the Composer's code generation process. The available macros are described in Table XIX.

3.2.4. SharedCode

The Supervisor has access to the Graph Type's SharedCode section from the XML. If it exists in the XML, it is placed at the top of the generated source file immediately below the properties and state declarations. SharedCode is not contained within a method and variables/functions declared in it may be used by other handlers. This is an ideal place to place `#includes` and `#defines` that are common throughout the entire application.

3.2.5. Code

The Supervisor's Code section from the XML is placed below the Graph Type's SharedCode. Code is not contained within a method and variables/functions declared in it may be used by other handlers. The Supervisor Code fragment may reference `supervisorProperties` and `supervisorState`. This is an ideal place to place `#includes` and `#defines` that are common throughout the Supervisor.

Table XIX: Supervisor convenience macros.

Macro	Evaluates to	XML handler	Usage
`GRAPHPROPERTIES(x)`	`Graph Properties->x`	all	Access to the Graph Properties struct. Usage is GRAPHPROPERTIES `(<MEMBER>) = <VAL>`
`SUPPROPERTIES(x)`	`Supervisor properties->x`	all	Access to the Supervisor Properties struct. Usage is SUPPROPERTIES `(<MEMBER>) = <VAL>`
`SUPSTATE(x)`	`Supervisor State->x`	all	Access to the Supervisor State struct. Usage is SUPSTATE `(<MEMBER>) = <VAL>`
`MSG(x)`	`message->x`	OnReceive	Access to member of the formatted payload struct for an inbound packet. Usage is `MSG(<MEMBER>) = <VAL>`
`PKT(x)`	`message->x`	OnReceive	Access to member of the formatted payload struct for an inbound packet. Usage is `PKT(<MEMBER>) = <VAL>`
`REPLY(x)`	`reply->x`	OnReceive	Access to member of the formatted payload struct for a reply packet. Usage is `MSG(<MEMBER>) = <VAL>`
`BCAST(x)`	`bcast->x`	OnReceive	Access to member of the formatted payload struct for a broadcast packet. Usage is `BCAST(<MEMBER>) = <VAL>`
`RTSREPLY()`	`__rtsReply= true`	OnReceive	Indicates that the handler has requested that the reply packet is sent.
`RTSBCAST()`	`__rtsBcast= true`	OnReceive	Indicates that the handler has requested that the broadcast packet is sent.

3.2.6. OnInit

A Supervisor's OnInit handler is called during initialisation. OnInit can be used to carry out any initialisation that cannot be handled with the generated properties and state initialisers.

OnInit is used to allocate `__SupervisorProperties` and `__SupervisorState` so it must always be called by the Mothership, even if no application-specific initialisation behaviour is defined. Calling OnInit multiple times has no effect as the first execution sets `__SupervisorInit = true` and all subsequent calls will return -1.

3.2.7. OnSupervisorIdle

`OnSupervisorIdle` may be called any time the Mothership has no other work to do. There is no guarantee when, or even if, this will be called.

3.2.8. OnImplicit

OnImplicit is called when the SupervisorCall() method is called and the packet has the `P_CNC_IMPL` opcode set. `OnImplicit` implements the `SupervisorInPin`'s `OnReceive` handler from the XML. In addition to the variables listed in Table XVIII, OnImplicit also has access to the variables listed in Table XX.

Table XX: Implicit receive handler local variables

Variable	Type	Description
`__inPkt`	`P_Pkt_t*`	A pointer to the input packet.
`__outPkt`	`std::vector<P_Addr_Pkt_t>&`	A reference to a vector of output packets and their destination hardware addresses.
`__reply`	`P_Pkt_t`	Packet used to contain a unicast reply.
`__bcast`	`P_Pkt_t`	Packet used to contain a broadcast.
`__rtsReply`	`bool`	Flag to indicate whether the reply packet should be sent.
`__rtsBcast`	`bool`	Flag to indicate whether the broadcast packet should be sent.
`message`	`const pkt_<MESSAGENAME>_pyld_t*`	Pointer to the payload of the input packet that triggered the handler. Exact type is application dependant.
`reply`	`pkt_<MESSAGENAME>_pyld_t*`	Pointer to the payload of the reply packet. Exact type is application dependant.
`broadcast`	`pkt_<MESSAGENAME>_pyld_t*`	Pointer to the payload of the broadcast packet. Exact type is application dependant.

The payload of the packet that triggered the handler is available through the `message` pointer (and the `MSG()` and `PKT()` macros). This is cast to the format specified for the Supervisor's implicit input pin in the XML. The source address of the packet can be determined from `DeviceVector[__inPkt->header.pinAddr]` if so desired.

3.2.8.1. Replies

A unicast reply may be sent to the device where the inbound packet originated from. The payload of this reply is available through the `reply` pointer (and the `REPLY()` macro). This is cast to the format specified for the Supervisor's implicit output pin in the XML. If the Supervisor does not have an implicit output pin, the format specified for the implicit input pin is used. A reply will only be sent if the handler sets `__rtsReply` to `true` (or calls `RTSREPLY()`).

After the user-supplied handler code has executed, the `__rtsReply` flag is checked. If it is `true`, a unicast packet with the `P_CNC_IMPL` opcode set is added to the `__outPkt` vector. The destination of this packet is determined by using the contents of the `pinAddr` header as an index into the `DeviceVector`.

3.2.8.2. Broadcasts

A broadcast may be sent to all devices hosted by the Supervisor. The payload of this reply is available through the `broadcast` pointer (and the `BCAST()` macro). This is cast to the format specified for the Supervisor's implicit output pin in the XML. If the Supervisor does not have an implicit output pin, the format specified for the implicit input pin is used. A broadcast will only be sent if the handler sets `__rtsBcast` to `true` (or calls `RTSBCAST()`).

After the user-supplied handler code has executed, the `__rtsBcast` flag is checked. If it is `true`, a packet with the `P_CNC_IMPL` opcode set is added to the `__outPkt` vector for each thread assigned to this supervisor, determined by iterating over the `ThreadVector`.

3.2.9. OnPkt

`OnPkt` is called when the `SupervisorCall()` method is called and the packet does not have the `P_CNC_IMPL` opcode set. Handler code has access to the same variables and packet sending mechanisms as the `OnImplicit` handler. This is currently a stub and is never populated.

3.2.10. OnRTCL

OnRTCL should be called in response to RTCL kicking the Supervisor. Currently unimplemented.

3.2.11. OnStop

OnStop is called to shut down the Supervisor and execute any application-specific shutdown code.

OnStop is used to free `__SupervisorProperties` and `__SupervisorState` so it must always be called when terminating execution, even if no application-specific shutdown behaviour is defined. Calling OnStop multiple times has no effect as the first execution sets `__SupervisorInit = false` and all subsequent calls will return -1.

N.B. `__SupervisorProperties` and `__SupervisorState` are not freed until after the user-supplied code has been executed - the user-supplied code MUST NOT include a return statement.

3.3. Supervisor Data Structure

The Supervisor's data structure consists of a bool to track initialisation state, a pointer to the Supervisor's state, a pointer to the Supervisor's properties, two vectors containing addresses and (temporarily) a `std::map` of device names and addresses.

The `DeviceVector` contains a `SupervisorDeviceInstance_t` for every device that the Supervisor is responsible for. A `SupervisorDeviceInstance_t` encapsulates the hardware and software addresses of a device. Temporarily (until the Nameserver is implemented) `SupervisorDeviceInstance_t` also includes the name of the device instance. The position (index) of a device in this vector corresponds to the index that a device includes in the `pinAddr` header when it sends to the Supervisor's implicit receive pin.

The `ThreadVector` contains the hardware address of every thread that hosts devices for the application. This may, but does not have to, contain the hardware address of every thread within the box. The ThreadVector exists so that the Supervisor can send packets to the implicit receive handler of every device that it is responsible for.

4. Compilation

The binaries for each of the cores involved in an application and the supervisor(s) can be compiled using make and the Makefile included with the Softswitch (at `Source/Softswitch/Makefile`). Currently, the build process will only work on Linux-based systems (including Windows Subsystem for Linux) that meet the pre-requisites listed in Section 4.2. The Softswitches and Supervisor shared object are compiled under the C++14 standard as the user-supplied C fragments may use features that are not supported under C++11 or C++98. In Orchestrator land, the user does not need to invoke this manually – the Composer handles the compilation.

4.1. Source Files

This section details the source and header files required for the Softswitch binaries and Supervisor shared object.

4.1.1. Softswitch

Each Softswitch binary (`softswitch_<CORE_ADDR>.elf`, and the derived instruction-space [`softswitch_code_<CORE_ADDR>.v`] and data-space [`softswitch_data_<CORE_ADDR>.v`] binaries) requires the source and header files listed in Table XXI. Each core will have up to 16 different `vars_<CORE_ADDR>_<THREAD_ADDR>.cpp` source files, one for each utilised thread.

Table XXI: Core source files. Filenames in italics are application and placement-specific.

Source	Header
`softswitch_main.cpp`	
`softswitch_common.cpp`	`softswitch_common.h`
`softswitch.cpp`	`softswitch.h`
`entry.S`	
	`poets_hardware.h`
`handlers_<CORE_ADDR>.cpp`	*`handlers_<CORE_ADDR>.h`*
`vars_<CORE_ADDR>.cpp`	*`vars_<CORE_ADDR>.h`*
`vars_<CORE_ADDR>_<THREAD_ADDR>.cpp`	
	`GlobalProperties.h`
	`MessageFormats.h`

4.1.1.1. Static sources

`softswitch_main.cpp/.h`: Contains the main entry point for the Softswitch.
`softswitch_common.cpp/.h`: Contains all of the common Softswitch routines.
`softswitch.cpp/.h`: Contains the trivial log handler.
`entry.S`: ASM routine to jump to main.
`poets_hardware.h`: Definitions of convenience.

4.1.1.2. Generated Sources

`handlers_<CORE_ADDR>.cpp/.h`: Contains the assembled user-supplied device handlers for the specified core. All devices on a core share handlers.
`vars_<CORE_ADDR>.cpp`: Contains the per-core variable definitions.
`vars_<CORE_ADDR>.h`: Contains declarations for the per-core properties and state and the per-thread variables.
`vars_<CORE_ADDR>_<THREAD_ADDR>.cpp`: Contains the per-thread variable definitions.
`GlobalProperties.h`: Contains the Global Properties declaration.
`MessageFormats.h`: Contains struct declarations for all of the message (packet) formats.

4.1.2. Supervisor

The Supervisor shared object requires the source and header files listed in Table XXII. *GlobalProperties.h* and *MessageFormats.h* are the same as used by the Softswitch.

Table XXII: Supervisor source files. Filenames in italics are application and placement-specific.

Source	Header	Other
`Supervisor.cpp`	`Supervisor.h`	
	`SupervisorApi.h`	
	`SupervisorApiEntrypoints.h`	
`supervisor_generated.cpp`	*`supervisor_generated.h`*	`supervisor.bin`
	`GlobalProperties.h`	
	`MessageFormats.h`	

4.1.2.1. Static Sources

`Supervisor.cpp`: Contains the shared object entry points that the Mothership uses to interact with the Supervisor.

`Supervisor.h`: Contains the declaration of the Supervisor class.

`SupervisorApi.h`: Contains the declaration of the SupervisorApi class.

`SupervisorApiEntrypoints.h`: Contains the entry point methods that an application may use to access the Supervisor API.

4.1.2.2. Generated Sources

`supervisor_generated.cpp/.h`: Contains definitions for the members of the Supervisor class. The definitions are assembled from parameters and handlers sourced from the XML.

`supervisor.bin` Contains the binary blob that holds the generated contents of the `DeviceVector`.

`GlobalProperties.h`: Contains the Global Properties declaration.

`MessageFormats.h`: Contains struct declarations for all of the message (packet) formats.

4.2. Prerequisites

In addition to the source files detailed in Section 4.1, several additional prerequisites are required to compile the Softswitch binaries and Supervisor shared object(s) with the provided Makefile:

Softswitch:

- poets_pkt.h (Source/Common/)
- OSFixes.hpp (Generics)
- SoftwareAddressDefs.h (Source/Common/)
- Tinsel 0.8
- RISC-V GNU toolchain configured for the rv32imf target architecture[11] [12].

Supervisor:

- macros.h (Generics)
- poets_pkt.h (Source/Common/)
- OSFixes.hpp (Generics)
- Tinsel 0.8
- mpich[11]

[11] Included in the Orchestrator dependencies tarball.

[12] The Makefile requires the toolchain to have the "riscv32-unknown-elf" prefix. Compilation may be possible with the 64-bit version of the toolchain.

MPICH is used as the C++ compiler for the Supervisor shared object as it will be present on any system running the Orchestrator. The Supervisor itself does not depend on MPI and may be compiled with an alternative C++ compiler.

4.3. Paths and Environment Variables

The Makefile uses the environment variables listed in Table XXIII to set paths that are involved in the compilation of the Softswitch and Supervisor binaries.

Table XXIII: Makefile environment variables and default paths.

Environment Variable	Default Path	Overloadable?	Description
TINSEL_ROOT	../../Tinsel	no	Location of the local Tinsel install or a symlink to it.
SOFTSW_ROOT	../../Softswitch	no	Location of the Softswitch common source or a symlink to it.
ORCH_ROOT	../../Orchestrator	no	Location of the Orchestrator or a symlink to it.
RISCV_PATH	/usr/local/riscv	yes	Location of the RISC-V GNU Toolchain or a symlink to it.
MPICH_PATH	/usr	yes	Location of the local MPICH install or a symlink to it.
GENDIR	../Generated	yes	Location of the generated Softswitch and Supervisor source.
BINDIR	../bin	yes	Location where output binaries are placed.

4.4. Configuration options

The build can be controlled by several command line arguments as detailed in Table XXIV. These are passed as arguments to `make`, e.g. to enable the buffering Softswitch and pass custom optimisation flags you would execute:

```
make SOFTSWITCH_BUFFERING=1 CM_FLAGS="-O3 -fno-inline"
```

Table XXIV: Makefile command line arguments to control Softswitch compilation.

Command line argument	Description
`SOFTSWITCH_BUFFERING=1`	Compiles the Softswitch with the `BUFFERING_SOFTSWITCH` definition set.
`SOFTSWITCH_DISABLE_INSTRUMENTATION=1`	Compiles the Softswitch with the `DISABLE_SOFTSWITCH_INSTRUMENTATION` definition set.
`SOFTSWITCH_TRIVIAL_LOG_HANDLER=1`	Compiles the Softswitch with the `TRIVIAL_LOG_HANDLER` definition set.
`SOFTSWITCH_PRIORITISE_INSTRUMENTATION=1`	Compiles the Softswitch with the `SOFTSWITCH_PRIORITISE_INSTRUMENTATION` definition set.
`SOFTSWITCH_LOGLEVEL=<LEVEL>`	Sets the logging level of the Softswitch to `LEVEL` by setting the `P_LOG_LEVEL` definition.
`CM_CFLAGS="<FLAGS>"`	Compiles the Softswitch and Supervisor with the additional compilation `FLAGS`. These are place after (and thus override) the default flags detailed in Section 4.5.

4.5. Default flags

Excluding library includes, the default flags passed to the RISCV compiler to build the Softswitch objects is:

```
-mabi=ilp32 -march=rv32imf -static -mcmodel=medany -fvisibility=hidden
    -nostartfiles -pipe -fsingle-precision-constant -fno-builtin-printf
    -ffp-contract=off -std=c++14 -Wall -O2
```

Excluding library includes, the default flags passed to the RISCV linker to create the Softswitch binaries is:

```
-melf32lriscv -G 0 -lgcc -lc
```

Excluding library includes, the default flags passed to `mpicxx` to compile the Supervisor shared object are:

```
-std=c++14 -fPIC -pipe -Wall -shared -O3 -Wl,-soname,libSupervisor.so
```

4.6. Linker Scripts

When compiling under GCC, each core requires a linker script to map the correct areas of the shared memory to the regions created in the binaries. A shell script for generating per-core linker scripts for the Tinsel cores is included with the Softswitch (Source/Softswitch/genld.sh). `genld.sh` is called automatically for each core as part of the make. `genld.sh` is called with the core number as the first argument and the output may be piped to the destination linker file. i.e. as: `./genld.sh 1 > link_1.ld`.

Alternative backends that make use of an unmodified Softswitch must provide a linker script that creates a section for the ThreadContext and variable that references the start of the section. An example that augments the default GCC linker script with a section for the Thread 0 ThreadContext is shown in Insert 1. This can be used during a link with GCC by adding `-Wl,-T<FILENAME>.ld` to the arguments of the link command.

Insert 1: Example Linker script to add a section for the Thread 0 ThreadContext.

```
SECTIONS
{
    .thr0_base_outer : {
        thr0_base_start = .;
        *(.thr0_base)
        thr0_base_end = .;
    }
}
INSERT AFTER .rodata
```

5. Composer

The Composer is responsible for assembling the Softswitch and Supervisor source files for an application. It is also responsible for initiating application compilation and for managing the output directory for source files and compiled binaries. `OrchBase` holds a Composer member at `OrchBase::pComposer`. The lifetime of `OrchBase::pComposer` is tied to the lifetime of `OrchBase::pPlacer`: anytime `OrchBase` replaces its `OrchBase::pPlacer` member, `OrchBase::pComposer` is also replaced.

This chapter provides details on the internal workings of the Composer and how it interacts with the rest of the Orchestrator and the operator.

5.1. Commands

Operator interaction with the Composer is facilitated through the commands detailed in Table XXV.

Table XXV: Composer commands for Operator interaction.

Command	Clause	Parameter(s)	Description
compose	/generate	= APPNAME	Generates source code[13].
	/compile		Compiles the binaries and Supervisor shared object(s)[13].
	/app		Equivalent to a /generate followed by a /compile.
	/bypass		Bypasses the generation and compilation steps and allows previously compiled binaries to be reused[13].
	/clean		Cleans the compiled binaries[13]. Effectively a "make clean".
	/degenerate		Cleans up the generated source code[13]. Removes all traces of the application from Composer.
	/decompose		Equivalent to a /clean followed by a /degenerate.
	/buff		Sets the Buffering Softswitch flag[13]. [14]
	/nobuff		Clears the Buffering Softswitch flag[13]. [14]
	/inst		Sets the Softswitch Instrumentation flag[13]. [14]
	/noinst		Clears the Softswitch Instrumentation flag[13]. [14]
	/args	= APPNAME, ARGS	Pass additional arguments to the compilers. See Sections 4.4 & 5.5.1.7 for more information.
	/logh	= APPNAME, LOGHANDLER	**Sets the Softswitch Log Handler for the[13]. See** Table XXVI for valid log handlers. [14]
	/logl	= APPNAME, LEVEL	Sets the Softswitch log level[13]. [14]
	/rtsb	= APPNAME, RTSBUFFSIZE	Sets the maximum Softswitch RTS/packet buffer size[13]. [15]
	/reset	-	Deletes existing Composer & creates a new one.
	/dump	-	Dumps diagnostic data to the microlog.

[13] for the specified application(s).
[14] Calls `clean` if the application has already been compiled.
[15] Calls `clean` if the application has already been compiled and degenerates the source.

Table XXVI: Valid parameters for the `logh` command

Log Handler	Description
`none`	No log handler is present. Useful for reducing binary size.
`trivial`	Enables the Softswitch's trivial log handler.

For these commands, `APPNAME` can have three forms:

- "*" (as in, just an asterisk), which performs the operation on all application graph instances in the Orchestrator.
- `APP` (as in, the name associated with an `Apps_t` instance), which performs the operation on all application graph instances associated with that application object.
- `APP::GRAPH`, which performs the operation on exactly one application graph instance.

These commands are implemented in `CmComp.cpp`.

5.2. Default Values

Unless changed by an Operator command, the Composer uses the default values shown in Table XXVII when generating and compiling the Softswitch and Supervisor.

Table XXVII: Default composer values

Softswitch build variable	Default value	Equivalent command
Log Handler	trivial	`/logh = *,"trivial"`
Log Level	2	`/logl = *,"2"`
RTSBuf max size	4096	`/rtsb = *,"4096"`
Mode	Non-buffering	`/nobuff = *`
Instrumentation	Enabled	`/inst = *`

5.3. Data Structure

The Composer's data structure is anchored around a private map of `GraphI_t` pointers to `ComposerGraphI_t` struct pointers:

```
std::map<GraphI_t*, ComposerGraphI_t*> Composer::graphIMap
```

Each `ComposerGraphI_t` tracks the code generation and compilation status of a single Graph Instance. A `ComposerGraphI_t` contains configuration information that affects the code generation and compilation along with cached data (such as common strings used by a large number of devices) that is used to improve code generation efficiency. The members of a `ComposerGraphI_t` are described in Table XXVIII.The Composer operates directly on the members of each instance of the `ComposerGraphI_t` struct: aside from a method to clear the `devTStrsMap`; a `Dump()` method and constructors/deconstructor, a `ComposerGraphI_t` does not offer any methods to interact with it.

The Composer's `generate()`, `compose()` and `setX/addX` methods create a new `ComposerGraphI_t` on the heap and add it to the add an entry for it to `graphIMap` if the `GraphI_t` being operated on has not been seen before. A `ComposerGraphI_t` is `deleted` when a `decompose()` (or a `degenerate()` with the second argument set to `true`) is called for the application. The Composer's deconstructor `delete`s any remaining `ComposerGraphI_t`s.

The `setX`/`addX` methods change options regarding the compilation of the binaries and/or generation of the source code. These methods `clean()` and `degenerate()` (without removing the application from the Composer's data structure) the application as appropriate. To avoid pointless compute, any required `setX`/`addX` methods should be called before the application is generated.

The Composer does not store the complete generated source or the compiled binaries within its data structure. These are stored in the specified output directory for the application. Any changes to code generation parameters require that all of the code is regenerated from scratch.

Table XXVIII: `ComposerGraphI_t` member list

Member	Description
`GraphI_t* graphI`	A pointer to the graph instance of interest.
`std::set<P_core*>* cores`	A pointer to the set of cores used by the application. This is sourced from the Placer.
`devTStrsMap_t devTStrsMap`	A map of device type pointers to a collection of strings. This map is populated during Generation and is used to cache strings that are common across multiple generated source files. As a typical application will generally have a small number of device types, this map will be relatively small.
`std::vector<DevI_t*> supevisorDevIVect`	A vector of device instance pointers for all devices serviced by the Supervisor. This is used to generate to generate the Supervisor's `DeviceVector`.
`devISuperIdxMap_t devISuperIdxMap`	A map of device instance pointers to a `uint32_t` representing the index of the device in the `supevisorDevIVect`. This is used to generate the `pinAddr` header for implicit output pins.
`std::string outputDir`	The full output directory where the generated source code and binaries will be placed. This is a concatenation of the Graph Instance's compound name and the Composer's output directory at the time the Graph Instance was first seen.
`std::string provenanceCache`	A cache of the information that is written at the start of every generated source file.
`std::string compilationFlags`	Additional (operator-supplied) flags that are passed to the Makefile verbatim.
`bool generated`	Flag that indicates the generated source files for the application have been compiled.
`bool compiled`	Flag that indicates the source files have been generated for the application.
`bool bufferingSoftswitch`	Flag to indicate whether the Softswitch for the application should be compiled in buffering or non-buffering mode.
`unsigned long rtsBuffSizeMax`	User-supplied override for the maximum size of the RTS list/Packet buffer.
`bool softswitchInstrumentation`	Flag to indicate whether the Softswitches for the application should be compiled with instrumentation enabled or not.
`ssLogHandler_t softswitchLogHandler`	Indicates the log handler that the application's Softswitch should use.
`unsigned long softswitchLogLevel`	Indicates the maximum level of message that the application's Softswitch will ignore.
`ssLoopMode_t softswitchLoopMode`	Indicates the loop mode that the application's Softswitch will operate in. Currently not settable by operator command.

5.4. Output Directory

The Composer is responsible for managing the directories where generated source code and compiled binaries for applications are placed. The Composer must be provided with an output directory that contains the symlinks/copies listed in Insert 2.

The output directory path must be provided by the Orchestrator after Composer has been initialised and may be changed at any time. the output directory does not change the storage location for application that have already been seen by the Composer, the only way to change this is to `degenerate` or `decompose` the application and then re-`compose` it.

Insert 2: Output directory structure

```
├── Makefile -> $(Orchestrator)/Source/Softswitch/Makefile
├── Orchestrator
│   ├── Generics -> $(Orchestrator)/Generics
│   └── Source
│       └── Common -> $(Orchestrator)/Source/Common
├── Softswitch -> $(Orchestrator)/Source/Softswitch
├── Supervisor.cpp -> $(Orchestrator)/Source/Supervisor/Supervisor.cpp
├── Supervisor.h -> $(Orchestrator)/Source/Supervisor/Supervisor.h
├── SupervisorApi.h -> $(Orchestrator)/Source/Supervisor/SupervisorApi.h
├── SupervisorApiEntrypoints.h -> $(Orchestrator)/Source/Supervisor/SupervisorApiEntrypoints.h
└── Tinsel -> $(TINSEL_ROOT)
```

5.5. Composer Internals

The Composer is realised as a single class (Figure 8) in `Source/OrchBase/Composer.cpp/.h`. The class provides 19 public methods, two constructors and a destructor to interact with the rest of the Orchestrator. The public methods are listed and described in Table XXIX.

The command set detailed in Table XXV, with the exception of the `reset` command, is mapped directly onto a subset of the public methods: with the exception of setting the buffering and instrumentation modes, there is a 1-1 mapping of command to public method.

Private methods prefixed with "`write`" write boiler plate code and code fragments to the generated source files. Private methods prefixed with "`form`" generate strings that are cached within `ComposerGraphI_t` for use by multiple write methods.

The current implementation of the Composer uses string/file streams with static strings defined in the Composer source code to generate the code for the Softswitches and Supervisors. While not extensible in itself, the Composer is implemented as a set of well segmented methods so that transition to alternative methods (e.g. some form of templating) will be relatively simple.

Figure 8: Composer class diagram.

Table XXIX: Summary of Composer public methods.

Method	Description
`int compose(GraphI_t*)`	Composes the specified graph instance. Equivalent to a call to `generate()` followed by a call to `compile()`.
`int generate(GraphI_t*)`	Generates the Softswitch and Supervisor source code for the specified graph instance if the graph instance has not already been generated.
`int compile(GraphI_t*)`	Compiles the Softswitch binaries and the Supervisor shared object for the specified graph instance if the graph instance has not already been generated. Will not execute unless the graph instance has been generated.
`Int bypass(GraphI_t*)`	Skips code generation and compilation for the specified graph instance, allowing previously compiled binaries to be reused. Checks that all required binaries exist.
`int decompose(GraphI_t*)`	The opposite to a `compose()` for the specified graph instance. Equivalent to a call to `clean()` followed by a call to `degenerate(…, true)`.
`int degenerate( GraphI_t*, bool)`	Removes any generated files and clears internal caches. If the second parameter is `true`, `degenerate()` removes all traces of the specified graph instance from the Composer's data structure. Will not execute if the graph instance has been compiled.
`int clean(GraphI_t*)`	Cleans any compiled binaries and compilation by-products for the specified graph instance by calling a `make clean` in the output directory.
`void setOutputPath( std::string)`	Sets the common output path that will be used for all generation and compilation artefacts.
`void setPlacer(Placer*)`	Sets the underlying placer used for sourcing placement information. This invalidates all graph instances seen by the Composer so results in the `graphIMap` being cleared.
`int setBuffMode( GraphI_t*, bool)`	Sets the buffering mode of the Softswitch for the specified graph instance. Calls `clean()` if it has already been compiled.
`int setRTSSize(GraphI_t*, unsigned long)`	Sets the maximum size of the Softswitch's RTS Buffer for the specified graph instance. Calls `clean()` if it has already been compiled and `degenerate(…,false)` if it has already been generated.
`int enableInstr(GraphI_t*, bool)`	Sets the Softswitch instrumentation state for the specified graph instance. Calls `clean()` if compiled.
`int setLogHandler(GraphI_t*, ssLogHandler_t)`	Sets the Softswitch Log Handler for the specified graph instance. Calls `clean()` if compiled.
`int setLogLevel(GraphI_t*, unsigned long)`	Sets the Softswitch log level for the specified graph instance. Calls `clean()` if compiled.
`int setLoopMode(GraphI_t*, ssLoopMode_t)`	Sets the Softswitch loop mode for the specified graph instance. Calls `clean()` if compiled.
`int addFlags(GraphI_t*, std::string)`	Add additional compilation flags to the Softswitch build. Calls `clean()` if compiled.
`bool isGenerated(GraphI_t*)`	Returns the generation state for the specified graph instance.
`bool isCompiled(GraphI_t*)`	Returns the compilation state for the specified graph instance.
`void Show(FILE * = stdout)`	Outputs a summary of the Composer's state to the specified FILE.
`void Dump(unsigned = 0, FILE * = stdout)`	Outputs a detailed account of the Composer's state to the specified FILE.

5.5.1. Build Control

The methods that influence code generation (`setRTSSize()`) and compilation (`setBuffMode()`, `enableInstr()`, `setLogHandler()`, `setLogLevel()`, `setLoopMode()` and `addFlags()`) may be called at any time but will either `clean()` and/or `degenerate()` as required. Where possible these methods should be called before an application is generated or compiled. These methods may be called before the Compose has seen the application, in which case they will add the application to the Composer's `graphIMap`.

5.5.1.1. setRTSSize

This method sets the maximum size of the Softswitch's RTS buffer in normal mode, or the size of the packet buffer in buffering mode, by setting the value of `rtsBuffSizeMax` in the application's `ComposerGraphI_t`. The method's second parameter is the size of the buffer in number of entries, not bytes.

5.5.1.2. setBuffMode

This method sets the `bufferingSoftswitch` flag in the application's `ComposerGraphI_t` to indicate whether the Softswitch should be built in buffering mode or non-buffering mode. Passing true as the second parameter sets the flag and results in the Softswitch being compiled in buffering mode.

5.5.1.3. enableInstr

This method sets the `softswitchInstrumentation` flag in the application's `ComposerGraphI_t` to indicate whether the Softswitch should be built with instrumentation enabled. Passing true as the second parameter sets the flag and results in the Softswitch being compiled in with instrumentation enabled.

5.5.1.4. setLogHandler

This method sets the value of `softswitchLogHandler` in the application's `ComposerGraphI_t` to indicate which implemented log handler should be used. The second argument is an enum with the possible values shown in Insert 3.

Insert 3: Valid values for the Log Handler enum.

```
typedef enum softswitchLogHandler_t
{
    disabled = 0,
    trivial
} ssLogHandler_t;
```

5.5.1.5. setLogLevel

This method sets the minimum level at which a message will be send by `handler_log()` by setting the value of `softswitchLogLevel` in the application's `ComposerGraphI_t`.

5.5.1.6. setLoopMode

This method sets the value of `softswitchLoopMode` in the application's `ComposerGraphI_t` to indicate which implemented loop mode should be used. The second argument is an enum with the possible values shown in Insert 4.

Insert 4: Valid values for the Softswitch Loop Mode enum.

```
typedef enum softswitchLoopMode_t
{
    standard = 0,
    priInstr
} ssLoopMode_t;
```

5.5.1.7. addFlags

This method appends the provided string to the `compilationFlags` field in the application's `ComposerGraphI_t`. This method may be called multiple times to add additional flags. During a compilation, the contents of `compilationFlags` are passed in with the `CM_FLAGS` variable with no modification.

The Composer does not introspect the provided flags: this means that they are not validated at all. It is also not possible to remove a flag once it has been set, short of decmonposing/degenerating the application with the `del` flag set.

5.5.2. Code Generation

For current single-supervisor behaviour, the high-level operation of the Composer during a Generate operation is:

- Prepare application output directory and subdirectories
- Interrogate Placer for a set of Cores that are involved in the application.
- Iterate through Device Types to form a cache of common strings for each type.
- Generate the file provenance information
- Generate the global graph properties header (`GlobalProperties.h`)
- Generate the message formats header (`MessageFormats.h`)
- Generate the binary blob, source and header for the Supervisor (`supervisor.bin`, `supervisor_generated.{cpp|h}`)
- Iterate over the set of cores and generate the per-core source and headers for each core (vars_<CORE_ADDR>.{cpp|h}, handlers_<CORE_ADDR>.{cpp|h})
    - Iterate over each thread contained by the core and generate the per-thread source (vars_<CORE_ADDR>_<THREAD_ADDR>.cpp) and appends to the per-core variables header (vars_<CORE_ADDR>.h)

5.5.2.1. Application output directory preparation

The composer stores all generation, build and compilation artefacts for an application in an application-specific sub directory under the currently configured output directory. The name of the application output directory is derived from the Graph Instance's compound name (`<APP_NAME>::<GRAPH_NAME>`) with the double-colon being replaced with a "__". The application output directory is transient (it is purged any time a `generate` is called) and no additional files should be placed under it.

The composer prepares the output directory for the application by first removing the directory if it already exists. The Composer then creates the subdirectory structure shown in Insert 5 and described in Table XIX.

Finally, the static sources for the Supervisor are copied to the Generated subdirectory and a copy of the Softswitch's Makefile is placed in the Build subdirectory.

Insert 5: Application output directory structure.

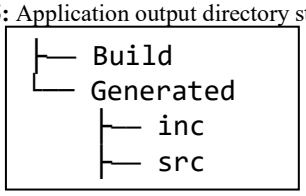


Table XXX: Application output directory layout

Directory	Description
Build	Build is initiated from here. Object files are placed here as part of the compilation process. A copy of the Makefile is placed here.
Generated	Contains copies of Supervisor source. Generated Supervisor source code, Supervisor binary blob, `GlobalProperties.h` and `MessageFormats.h` are placed here.
Generated/inc	Softswitch-specific headers (`handlers_%,h` and `vars_%.h`) are placed here.
Generated/src	Softswitch-specific source files (`handlers_%.cpp`, `vars_%.cpp` and `vars_%_%.cpp`) are placed here.

5.5.2.2. Device type strings formation

All devices of the same type share a large number of common strings throughout their generated source code. These strings include common preamble used at the start of every handler and the contents of the handler source and header files. To improve generation speed, each instance of a `ComposerGraphI_t` stores a cache of common strings (in the form of a map with the signature `std::map<DevT_t*, devTypStrings_t*> devTStrsMap`) for each device type involved in the application.

5.5.2.3. File provenance information

The Composer places a multi-line comment at the top of each generated source file that contains provenance information that provides details when the file was generated, the origin XML used to generate the source and Orchestrator/Composer settings at the time the file was generated. The provenance information is generated once and cached within the `ComposerGraphI_t`.

5.5.2.4. Global Properties header

The Global Properties header (`GlobalProperties.h`) includes the definition for the `global_props_t` struct and the declaration of the const-protected `GraphProperties` variable. This header is used by the Supervisor and all Softswitches. The file is created even if the application does not have any global properties.

5.5.2.5. Message Formats header

The Message Formats header (`MessageFormats.h`) contains structure definitions for each packet format used within the application. Packet formats structs are given generated names `pkt_<PACKETID>_pyld_t` where PACKETID is the contents of the "id" attribute from the XML. The file is created even if the application does not have any defined packet formats.

5.5.2.6. Supervisor Generation

When generating the Supervisor, the Composer walks every node in the Graph Instance's application digraph and populates the Supervisor's `DeviceVector` with device addresses. To significantly reduce the compilation time of the Supervisor for larger problems compared to writing a very large braced initialiser or initialisation method, the `DeviceVector` generated by the Composer are written to a binary file. This file is subsequently turned into a linkable object using `ld`'s `--relocatable`

option to partially link the file. This produces an object that contains symbols for pointers to the start and end of the binary blob, which can then be used to initialise the `const`-protected `DeviceVector`.

The digraph walk is also used to populate the `ComposerGraphI_t`'s `supevisorDevIVect` vector and `devISuperIdxMap` map. The Composer also walks the set of cores involved in the application to populate the Supervisor's `ThreadVector`. Finally, the Composer writes the Macros of Convenience and remainder of the Supervisor source code to `supervisor_generated.cpp` and `supervisor_generated.h`.

5.5.2.7. Softswitch Generation

To generate the Softswitch, the Composer starts by iterating over the set of cores involved in the application. For each core, the Composer first determines the device type hosted on the core by making use of the fact that a core only hosts a single device type and that if a core appears in the set of cores, at least one thread on the core (thread 0) hosts at least one device. The Composer then creates and populates the source and header files for the per-core handlers and variables (`handlers_<CORE>.{cpp|h}`, `vars_<CORE>.{cpp|h}`). The Composer then iterates over each thread hosted by the core.

For each thread, the Composer first creates the per-thread source file (`vars_<CORE>_<THREAD>.cpp`) and writes the initialisers for the thread-level constructs (`ThreadContext`, the Device Type array and the input/output pin types) before iterating over each device instance hosted on the thread.

As part of the `ThreadContext` generation, the Composer determines the size of the RTS/packet buffer. In buffering mode, this is simply the value of `rtsBuffSizeMax` stored in the application's `ComposerGraphI_t`. In non-buffering mode, and assuming that the device type has an implicit Supervisor send pin, the size is set to:

```
1 + <number of connected pin types> + <number of devices on thread>
```

as long as this is less than the value of `rtsBuffSizeMax` stored in the application's `ComposerGraphI_t` and greater than `MIN_RTSBUFFSIZE` (default 10). Where the device type does not have an implicit Supervisor pin, the `<number of devices on thread>` term is omitted.

For each device instance, the Composer forms and writes initialisers for the device instance, device state and device properties arrays. As part of this process, the Composer finds all of the edges that involve the device and writes the declarations for the input pin and output pin arrays by iterating over the edges. If the device type has an implicit Supervisor output pin, the Composer adds this pin to the end of the output pin array.

5.5.3. Compilation

The `compile` command is only effective for applications that have already been generated. If `compile` is called for an application that has not been generated, the Composer returns with a non-zero method and writes to the micrologg. Calling `compile` on an application that has already been compiled has no effect beyond a micrologg message.

During compilation, intermediate build objects are placed in the `Build` subdirectory and compiled binaries are placed in `bin` subdirectory, which is created as part of the compilation process. The Composer then forms the command used to invoke the Softswitch's Makefile by interrogating the contents of the application's `ComposerGraphI_t`. The format of the make invocation is shown in Insert 6. The output of the command, including anything written to `stderr`, is written to `Build/make_errs.txt`.

Insert 6: Layout of the make command executed by the Composer. Optional parameters that are only passed if the relevant value is set in the application's `ComposerGraphI_t` are enclosed in square braces.

```
make -j$(nproc --ignore=4) all [SOFTSWITCH_BUFFERING=1]
   [SOFTSWITCH_DISABLE_INSTRUMENTATION=1] [SOFTSWITCH_TRIVIAL_LOG_HANDLER=1]
   [SOFTSWITCH_PRIORITISE_INSTRUMENTATION=1]
   SOFTSWITCH_LOGLEVEL=(softswitchLogLevel) [CM_CFLAGS="(compilationFlags)"] >
   make errs.txt 2>&1
```

Once the command has been formed, the Composer invokes make from the application's `Build` directory. A parallel make is invoked to speed up the compilation – it will use all virtual cores save for four to allow other processes to continue.

If the compilation fails, the contents of `Build/make_errs.txt` are copied to the Microlog and the composer returns a non-zero value.

If the compilation was successful, the Composer walks the set of cores involved in the application and ensures that the binaries for each core were generated successfully. The Composer also stores the path to the generated instruction and data binaries for each core within the relevant `pCore`. The Composer also checks that the Supervisor's shared object has been generated and stores the path to it in the Graph Instance's `pSupI`. If any of the binaries have not been generated, the Composer writes to the microlog and returns with a non-zero value.

5.5.4. Clean

`clean()`undoes a compilation by invoking `make clean` in the application's `Build` directory. The Composer then removed the `bin` directory before walking all of the cores involved in the application and removing the paths for the previously generated binaries. Finally, the path stored in the Graph Instance's `pSupI` is cleared.

5.5.5. Degeneration

`degenerate()` undoes the generation step for applications that have been generated but not compiled. If degenerate is called for an application that has been compiled, the Composer writes to the microlog and returns with a non-zero value. Calling `degenerate` on an application that not been generated has no effect beyond a microlog message.

To degenerate an application, the Composer deleted the application's output directory and all of its contents. The Composer then clears the generated flag; the Supervisor device vector and map; the provenance string cache; and the Device Type strings cache in the application's `ComposerGraphI_t`. If the degenerate method has been called with the `del`(ete) flag set to `true` (i.e. when called by the `degenerate` or `decompose` commands), the Composer deletes the application's `ComposerGraphI_t`.

5.5.6. Status Interrogation

The Orchestrator may query the build and generation state of an application with the `isCompiled()` and `isGenerated()` methods. Both methods take a pointer to a `GraphI_t` as their only parameter and both methods return a `bool`.

5.6. Error Handling

Methods within the Composer that may fail return an integer that is non-zero on failure. Any failure is fatal and errors are written to the microlog for further inspection. Partially generated source is not tidied up: it will be removed when `generate` is next called.

Errors encountered during compilation are written to `Build/make_errs.txt` and then copied into the microlog. Errors during a clean operation are written to `Build/clean_errs.txt`.

6. Future Work

This section contains a brain dump of future work and improvements that could be made to the Softswitch, Supervisor and Composer.

6.1. Softswitch

6.1.1. Datastructure Location

Currently, the Softswitch does not make use of the available SRAM space (~16K) and stores everything in DRAM. The ThreadContext is stored at the base of the DRAM heap and everything else (including the RTS buffer and packet buffer) is stored on the stack.

The Softswitch could make better use of the available resources by storing the ThreadContext (~100 bytes) at the base of the SRAM heap and the RTS buffer at the base of the DRAM heap, or by using the SRAM heap for the packet buffer (of 292 packets) when in buffering mode.

6.1.2. Softswitch Shutdown

The current shutdown procedure relies on the FPGA fabric is rebooted between subsequent runs to clear stale packets, reload the softcores, etc. The shutdown process could be improved to include the following procedure:

- Send packet to Mothership to confirm shutdown has started
- Enter a loop that continuously drains the network while waiting for a final "kill" packet from the Mothership and discarding all other packets.
- Once the kill packet has been received, the Softswitch could then re-enter the bootloader.

6.1.3. Task field

The current version of the Softswitch does not check or use the task field in the header. For a multi-task deployment, this is required. Checking the task field can also help mitigate a potential problem of delayed/stale packets from a previous run from interfering with a current run.

One potential method of doing this would be to add a field to the ThreadContext for the task and setting it during the barrier by sending the task in the payload of the INIT packet from the Mothership.

6.1.4. Log Handler Improvements

The current log handler calls a trivial log handler (that ignores any provided arguments) and does not include the source device address. While the trivial log handler should be retained to provide a low-overhead method of sending log packets, a more full-featured log handler (selectable at build time) is required for proper application debugging.

6.1.5. Instrumentation Output

Instrumentation is currently written to individual files for each thread in the execution directory (`~/.orchestrator/instrumentation`). This should ultimately end up at the LogServer or a dedicated instrumentation handling process.

6.1.6. RTSbuf improvements

The RTS list is currently implemented as a circular buffer. The intention is that it is sized so that every pin on every device could have a pending send simultaneously. As such, the available space in the buffer is not checked when in non-buffering mode.

For larger problems (e.g. >1024 devices or lots of output pins per device), having buffer space for every output pin will be an inefficient use of resources. A more efficient method would be to have a fixed-size buffer that limits the number of pending pins. However, this means that some form of available space checking will need to be carried out before adding output pins to the RTS list.

6.1.7. Additional Execution Order Configuration

The Softswitch currently supports reordering of the receive and instrumentation sending operations within the loop. This could be extended to allow sending to be prioritised over receive (and instrumentation to be prioritised over that)..

6.2. Supervisor

The current Supervisor makes limited use of the Supervisor API and does not support multiple-supervisor execution.

6.3. Composer

6.3.1. Multi-supervisor support

For multi-supervisor operation, the generation sequence will (when implemented) become:

- Prepare application output directory and subdirectories
- Interrogate Placer for a set of Boxes that are involved in the application.
- Iterate through Device Types to form a cache of common strings for each type.
- Generate the file provenance information
- Generate the global graph properties header (GlobalProperties.h)
- Generate the message formats header (MessageFormats.h)
- Iterate over the set of boxes to:
  - Generate the source and header for the box's Supervisor (supervisor_generated.{cpp|h})
  - Iterate over the cores contained by the box and generate the per-core source and headers (vars_<CORE_ADDR>.{cpp|h}, handlers_<CORE_ADDR>.{cpp|h})
    - Iterate over each thread contained by the core and generate the per-thread source (vars_<CORE_ADDR>_<THREAD_ADDR>.cpp) and appends to the per-core variables header (vars_<CORE_ADDR>.h)

6.3.2. Templating

The Composer could be made more flexible (for alternative platforms or Softswitch alternatives) by moving to a template-inspired generation process. In other words, move away from having strings for the boilerplate code baked into the Composer and have them stored in a (selectable) set of files.