FLIC Programmer's Guide and API

v1.2 / July 2018

Intel® Movidius™
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Revision History

Date	Version	Description
July 2018	1.2	Replaced “Myriad 2” with “Myriad” as this document addresses both Myriad 2 and Myriad X.
April 2018	1.1	Updated section 13.1 Plugin Interfacing Guidelines with regard to maintaining consistency in plugin naming.
January 2018	17.12 (1.0)	Initial version.

Contents

1 Myriad FLIC Framework......6

1.1 Intended Audience......6
1.2 Scope......6
1.3 Dependencies......6
1.4 Licensing Information......6
1.5 Disclaimer......6
1.6 Support Information......6
1.7 Referenced Documents......7
1.8 Glossary of terms......7

2 FLIC Overview......8

2.1 Rationale......8
2.2 General......9
2.3 Restrictions......9
2.4 Communication......9

3 Plugins......11

3.1 Example [Source plugin]......11
3.2 Example [Output plugin]......11
3.3 Example [ISP plugin]......12

4 Frame pools......13

4.1 Frame travel......13
4.2 Custom buffering......14
4.3 Allocators......15

5 Communication models (PUSH/PULL)......16

6 Logging interface......17

6.1 Pipeline operations......17
6.2 Message operations......18
6.3 Logging control......18

7 Return values......19

8 IO API......20

8.1 Master-Sender......20
8.2 Slave-Sender......20
8.3 Slave-Receiver......20
8.4 Master-Receiver......21
8.5 Timed operations......21

9 Pool API......22

10 Plugin API......23

11 Pipeline API......25

12 Inter Leon communication......26

12.1 RmtO->RMTI......26
12.2 Makefile options......27

12.3 Init/Destroy....27
12.4 Interrupts....27
12.5 Passing Frame Pointers....28

13 Plugin Guidelines....29

13.1 Plugin Interfacing Guidelines....29
13.2 Plugin Development & Test Guidelines....29
13.3 Inter Plugin Communication Guidelines....30

14 Miscellaneous....31

14.1 IO Conditions....31
14.2 Memory checks....31
14.3 Interrupts....32
14.4 Template code size per type....32
14.5 Dynamic allocation....32
14.6 RTEMS thread policies/suspicious hangs....33
14.7 Buffering policies....33
14.7.1 Queue....33
14.7.2 Overwrite oldest....33
14.7.3 Custom buffering example....34
14.8 Synchronized outputs....34
14.9 Adapter....34
14.10 Plugin-groups....35
14.11 Atomic section....35

15 Tests....36

1 Myriad FLIC Framework

This document is intended to be the Programmer's Guide for the FLIC framework (FLIC stands for "**F**rame **L**evel ISP **C**V").

1.1 Intended Audience

This document is intended for software engineers programming the Myriad family of devices. The goal is to familiarize them with the FLIC programming framework.

This document assumes the reader is familiar with the Myriad processor [1], the Myriad Development Kit (MDK) [2, 3] and the development board (mv0182 / mv0212) hardware platform [4][5].

1.2 Scope

This document describes the following:

- Supported software capabilities and features of FLIC.
- Build and install instructions.
- How to build and run the FLIC example applications.

1.3 Dependencies

WARNING: FLIC is built with the Movidius MDK v17.11 and tools release v0.87.3 The MDK and tools are delivered via www.movidius.org support portal.

1.4 Licensing Information

Use of this software is covered by the appropriate Intel Movidius Software License Agreement.

1.5 Disclaimer

The information in this document and any document referenced herein is provided for informational purposes only, is provided AS IS AND WITH ALL FAULTS and cannot be understood as substituting for customized service and information that might be developed by Intel Movidius for a particular user based upon that user's particular environment. RELIANCE UPON THIS DOCUMENT AND ANY DOCUMENT REFERENCED HEREIN IS AT THE USER'S OWN RISK.

1.6 Support Information

For MDK support you can post your questions on tickets at https://www.movidius.org/

1.7 Referenced Documents

No.	Document	Revision
1	Myriad MA2x5x DB-R	0.98
2	MDK Getting Started Guide	3.3
3	MDK Programmer's Guide	3.5
4	Movidius MV0182 User Manual	1.7
5	Movidius MV0212 User Manual	1.0
6	Ma2x5x SIPP User Guide	1.3
7	Ma2x8x SIPP User Guide	1.2

1.8 Glossary of terms

Term	Description
FLIC	**F**rame **L**evel **I**sp **C**v
ISP	**I**mage **S**ignal **P**rocessing
MIPI	**M**obile **I**ndustry **P**rocessor **I**nterface
USB	**U**niversal **S**erial **B**us
CV	**C**omputer **V**ision
RAW	Bayer image format
YUV	Image format consisting of one luma(Y) and two chroma (UV) components.
RGB	Image format consisting of 3 color components: Red, Green, Blue.
Plugin	Leon Control/processing block (container for Sender/Receivers).
Sender	Block that sends a message/frame to one or more Receivers.
Receiver	Block that receives messages from one or more Senders.
IO	**I**nput/**O**utput. In this document this refers typically to a Plugin's IOs which are Senders and Receivers.

2 FLIC Overview

2.1 Rationale

Given increasingly complex application scenarios (including ISP, CV, codecs, etc…) a common communication **control** and **data** sharing mechanism is required on Myriad.

Adopting a common interface will reduce integration time when building new scenarios/applications.

The FLIC framework provides the following functionality:

- Allows apps to instantiate multiple Plugins (i.e. the processing/control blocks).
- Allows apps to connect Plugins together to create a Pipeline by forming a graph.
- Applications may create multiple, independent Pipelines.
- Handle frame-buffers (pixel/metadata/control param buffers) via Pools & shared-pointer approach.
- User-configurable logging of message transfers.
- Allows messages to be buffered with customizable buffering policies, e.g.:
    - custom message READ: based on timestamp value (required when a slave unit has to sync with a master).
    - custom message WRITE: overwrite oldest-frame (required when slow consumers shouldn't block fast producers and other fast consumers of same data).

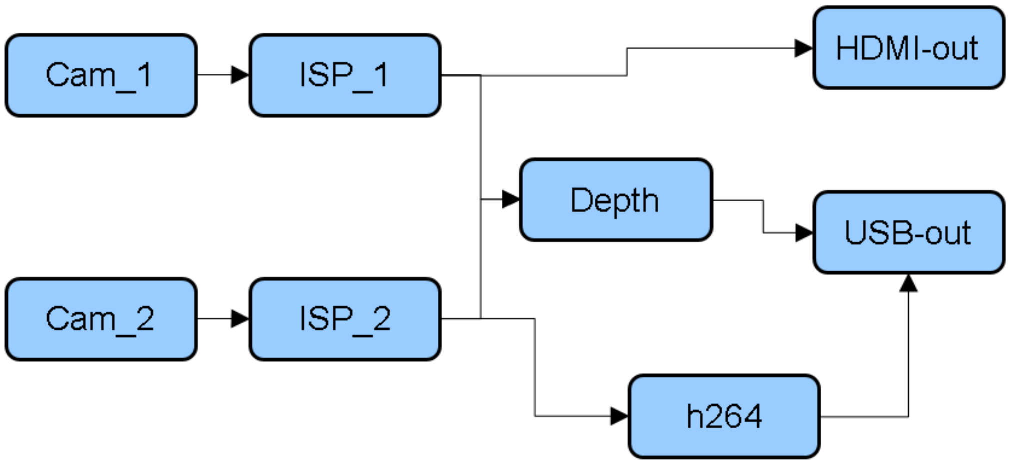


Figure 1: FLIC pipeline example

2.2 General

FLIC is a C++ RTEMS-based component that allows **frame-level pipelining** between multiple processing/control blocks called **Plugins**. **Plugins** are grouped in **Pipelines** and communicate to each other via Input/Output (IO) entities called **Sender** and **Receiver**.

A **Plugin** typically contains:

- zero-or-more **Receivers**.
- zero-or-more **Senders.**
- app specific data and logic/control.

An application can contain one or more FLIC pipelines.

- FLIC learning examples: `/mdk/common/components/flic/tests/`
- FLIC live reference apps: `/mdk/common/components/flic/examples/`

IMPORTANT:

FLIC Input/outputs are C++ templates. In order to connect two plugins together, types need to be compatible. In order to maintain a consistent & healthy plugin echosystem where various plugs can & should connect, programmers should try to adhere to commonly adopted types where possible:

- For image frames (e.g. RAW, YUV, RGB, MONO, ….), the ImgFrame type should be used.

2.3 Restrictions

FLIC adopted the following restrictions:

- Not using **std** library.
- Not using dynamic memory allocation at run-time (i.e. after pipeline Start).

2.4 Communication

Sender → Receiver communication is unidirectional and message buffering is typically done by Receiver.

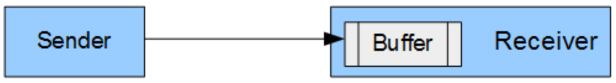


Figure 2: Receiver buffering

NOTE: There are some exceptions to this rule, when buffering is done at Sender side (see section 4, Frame pools).

Sender and **Receiver** blocks are defined as C++ templates.

Inserting a message into the message buffer is done via **copy-constructor**, whereas popping a message is done via **= operator** followed by invoking C++ destructor on the location that just got popped.

Communication granularity is typically frame-based or a few notifications per frame. It's not recommended to plan for inter-plugin notifications at line-group resolution (e.g. a notification each micro-seconds or so).

A **Sender** can push data towards multiple Receivers (that belong to different plugins) and each of the Receivers could implement different message buffering policies.

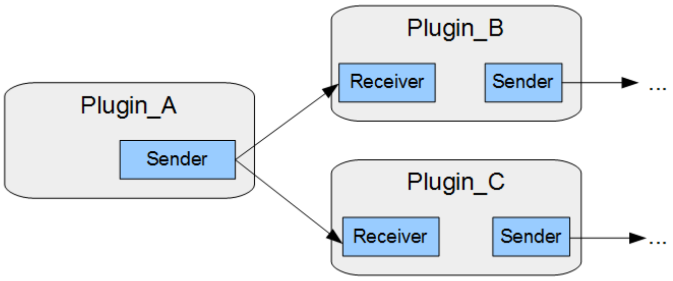


Figure 3: Multiple Receivers

It's also valid to connect multiple senders to same unique **Receiver** (e.g. assume multiple ISPs push data over an UsbOutput plugin to outside world).

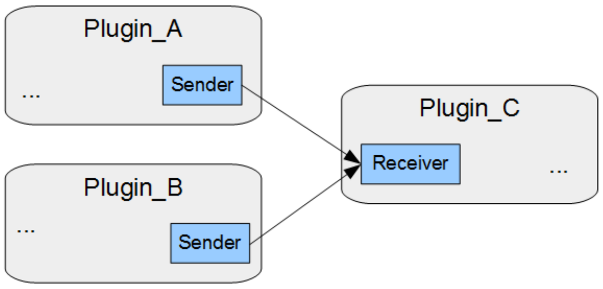


Figure 4: Multiple Senders, single Receiver

3 Plugins

The Plugin behavior/logic is implemented by the user via `ThreadedPlugin::threadFunc` method. Thus, the Plugin main body runs on Leon yet it can defer some of the responsibility to a Shave if required.

By default, each Plugin gets one thread associated, but the user can start/manage new threads if needed.

Most Plugins supervise execution of various HW blocks, but some Plugins are purely Leon oriented (e.g. filter/generate parameters, FramePool plugin).

The typical plugin operation is:

- Waits for IO conditions to be met.
- Kick processing and wait for a "done" event.
- Push results to consumers.

3.1 Example [Source plugin]

The [Source Plugin] is a wrapper for the MipiRx block/driver and has a single output, through which it informs consumers that new frames arrived from the Camera module.

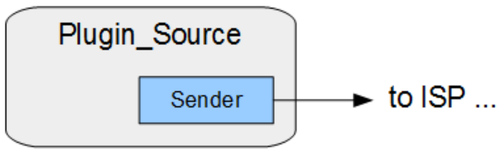


Figure 5: Basic Source

NOTE: In practice, Source has also a Receiver that connects to a FramePool, through which Source gets an available/reusable RAW-frame that the MipiRx block can write next.

3.2 Example [Output plugin]

This plugin receives data from another plugin and sends it over a physical channel (e.g. HDMI, USB, MipiTx).

The Output plugin has a single input (Receiver) and no outputs.

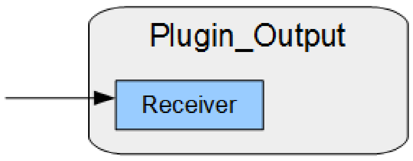


Figure 6: Basic Output

3.3 Example [ISP plugin]

The Image Signal Processing (ISP) plugin is a typical case of Plugin with multiple inputs and multiple outputs.

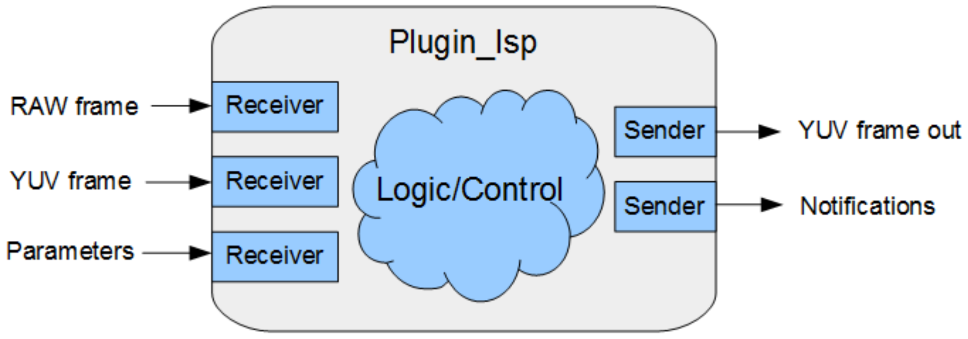


Figure 7: ISP plugin

Description:

- **RAW frame**: input a Bayer-RAW frame. This frame is typically received from parent plugin.
- **YUV frame**: a YUV buffer where plugin will write the results. This frame typically comes from a FramePool.
- **Parameters**: ISP specific parameters that are used to generate the output YUV frame.
- **Logic/Control**: is typically implemented via `ThreadedPlugin::threadFunc` method. This block typically makes sure all run conditions are met, and when they are, it kicks the underlying framework (e.g. SippFw, [6]) to execute the complex ISP processing.
- **YUV frame out**: when ISP processing is done, this output notifies all consumers that a new YUV frame is available.
- **Notifications**: contains multiple events that some blocks expect (e.g. ISP statistics are ready, ISP started, ISP finished, certain line is hit, etc…).

4 Frame pools

Frame pools are typically used to feed the Plugins that produce (i.e., write) frames. The producer Plugin requests an available frame from Pool (via a `Receive()` call) and after data is written in the frame buffer, that frame is typically forwarded to consumers via an `output.Send()` type of call.

An "available/free frame" is a frame that isn't currently referenced (i.e. used) by any Plugin.

Simply put, a **frame** is defined by a BASE and a SIZE [bytes] (see PoBuf definition), but the user can extend the descriptor associated to a frame by extending PoBuf (e.g. `flic/types/ImgFrame.h`).

NOTE: The frame producer/writer is responsible to update the frame descriptor appropriately before publishing to consumers (frame producer knows details such as frame-type, resolution, etc.).

A frame can hold: pixel-buffer, metadata, statistics-buffer, control/config parameters, etc.

Frame pools are available for users via stock **PlgPool** plugin. **PlgPool** has Slave-Sender output (named "out") through which it provides shared-pointer-like structures pointing to available free frames.

Frames within a frame-pool become referenced by a plugin when a **Receive** call (from that pool) succeeds.

The receiver Plugin gets a shared-pointer (see **PoPtr** class) that points to an available frame.

When the shared-pointer (PoPtr = Pool Ptr) goes out of scope, the frame gets released automatically.

The user can decide to manually release a frame by doing a **= nullptr** type of assignment or invoking the `PoPtr::Reset()` method.

When a frame is not referenced anymore, it returns to its originating pool.

4.1 Frame travel

The typical scenario for a plugin that needs an [Input frame] and writes an [Output frame] is illustrated below:

- Input frame typically comes from parent plugin and it's typically READ-ONLY.
- Output frame (i.e. the one that the current plugin is about to WRITE) typically comes from an associated Frame-Pool plugin.

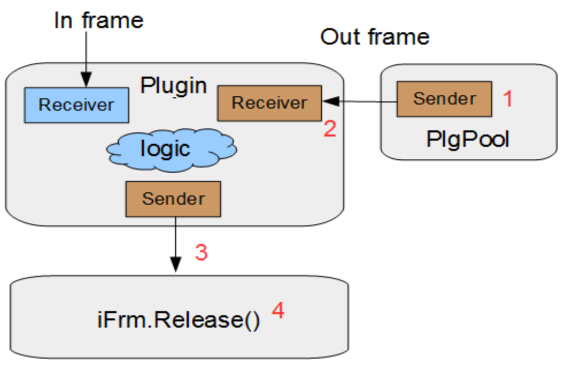


Figure 8: Frame travel

The progress of a Frame as it travels is described as follows:

T1: Frame is ready to use in framePool.

T2: Plugin received the input frame and now it needs an output frame where to place results. Plugin will now request a frame from plugin via "`inO.Receive(&oFrame);`".

T3: Plugin finished writing the output frame and informs consumer plugin via "`out.Send(&oFrame);`", then releases input and output frames as it doesn't need them anymore.

T4: Consumer plugin reads/consumes the frame(s) and eventually releases it, at which stage the frame becomes free and returns in the pool it came from.

NOTE: This concept is a bit contradictory: "*The output frame is an output, why do we need an input to receive it?*" Indeed, the output frame is an "output" in the sense that the current plugin does a WRITE operation on that buffer, but the buffer address that we can write is sometimes an external decision, therefore the address is an input.

Figure 9 is an example of how plugins and their associated pools are connected. The pipeline is made of 3 plugins plus 3 associated pools that feed their output.

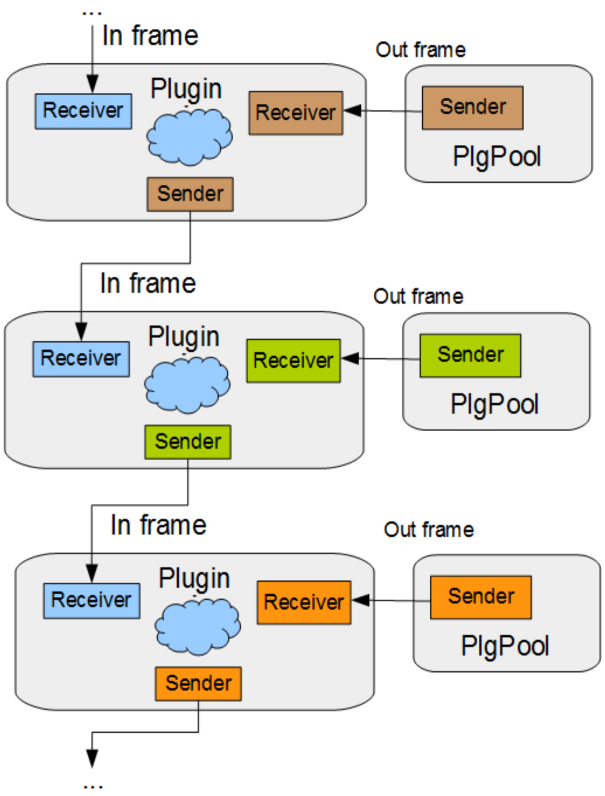


Figure 9: Multiple Plugs and Pools

4.2 Custom buffering

Having the output fed by another plugin allows integration with other blocks with special requirements. Figure 10 is an example of how an ISP plugin could integrate with a Video Encoder that manages its own input frames, but can also operate standalone (in conjunction with a frame-pool). Only relevant IOs are included.

Since Video Encoder manages its own input buffer (which is also the ISP output buffer), when ISP wants to write a new frame, it would need to ask the Video Encoder for a free frame. The Video Encoder plugin has a double role in this case:

- Pool for the ISP YUV frames.
- Consumer of the ISP YUV frames.

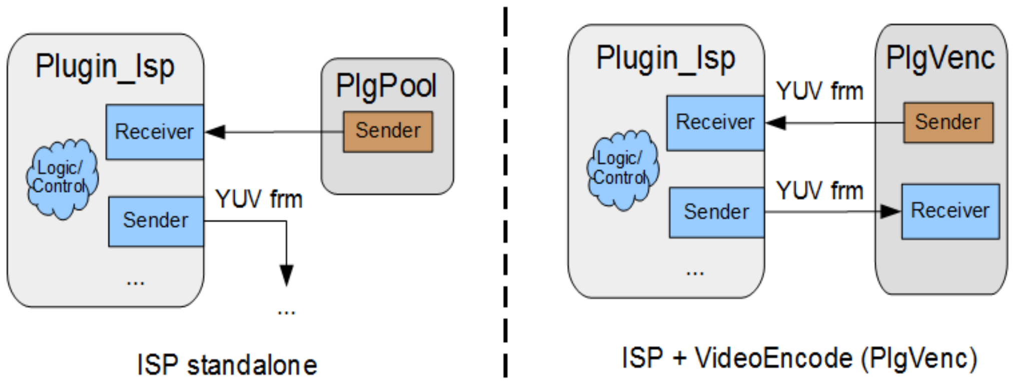


Figure 10: ISP integration

NOTE: First version of Myriad X VideoEncoder driver managed the encoding YUV buffers itself as shown in the right side view of Figure 10, but current driver operates as in the left side view of Figure 10. Other VideoEncoder HW/DRV solutions exist, where YUV buffers are managed from application space, in which case a regular PlgPool can be used to hold the YUV frames which are fed to HW encoder (Figure 10: left side view).

4.3 Allocators

Frame pools memory allocation can be customized by the user via Allocator object.

An Allocator object allows for Alloc/Free operations that users can define in custom ways.

By default, FLIC provides 2 Allocator objects:

1. **Heap allocator**: suitable for when the heap is mapped in DDR, see **HeapAlloc** object).
2. **Region allocator**: user can declare a static buffer anywhere in the memory system and use via RTEMS's region manager (see default **`RgnAllocator`** class). A default `RgnAllocator` object is provided as default (useful for tests purposes), see `RgnBuff` and `DEF_POOL_SZ`, `DEF_POOL_SECT` macros.

NOTE: By default, RgnBuff pool is tiny in order not to impact general applications. If the user wants to use it, the user needs to increase the pool size via Makefile options.

Users can customize memory allocation by implementing **IAllocator** interface.

5 Communication models (PUSH/PULL)

As mentioned before, message buffering is typically handled by the **Receiver** and this mode is used to implement **PUSH** schemes where the **Sender** pushes data at any time it wishes, and the **Receiver** is responsible to buffer the incoming messages.

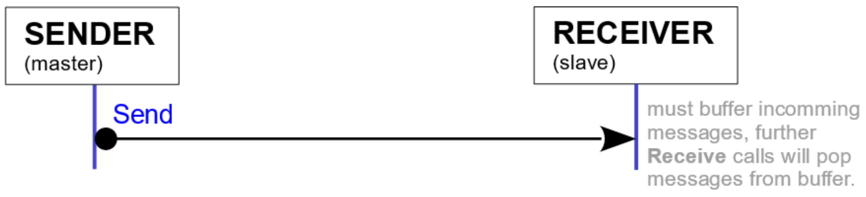

SENDER
(master)
RECEIVER
(slave)
Send
must buffer incomming messages, further Receive calls will pop messages from buffer.

Figure 11: PUSH model

But other times a **PULL** scheme is more appropriate (e.g. Frame-Pool, Myriad X VideoEncode), in which case the Sender does the message buffering/management, and only pushes data towards the Receiver on demand (i.e., when Receiver executes a `::Receive` call).

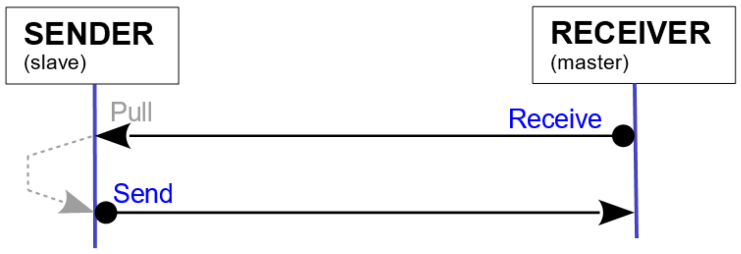


Figure 12: PULL model

The two PUSH/PULL paradigms are implemented via two types of objects:

- **PUSH**: Master_Sender + Slave_Receiver (see MSender, SReceiver classes).
- **PULL**: Slave_Sender + Master_Receiver (see SSender, MReceiver classes).

Basically, the MASTER is the block that initiates the transfer. A Slave can only connect to a Master and vice-versa.

Typically the FramePool and VideoEncode plugins contain a Slave-Sender, so they need to be matched with Master-Receivers.

Otherwise Master/Slave communication is Master-Send and Slave-Receive.

If one tries to connect incompatible Slave-Slave or Master-Master blocks, a compile-time error will be generated, so one will typically need to rename some data types from say **SReceiver** to **MReceiver**.

6 Logging interface

FLIC allows users to log its main operations via **Logging interface**. Programmers decide:

- How logging is implemented (e.g. using TraceAPI, printf, binary logs).
- What information is to be logged (starting from generic message type T).

NOTE: FLIC does not provide a full-blown ready-to-use logging infrastructure (like TraceAPI or mvLog), it only provides some interfaces/hooks to allow users to capture relevant FLIC events. Users should use full-blown logging APIs (e.g. TraceAPI or mvLog) to implement these interfaces.

NOTE: Even though users can customize how logging is implemented, it is recommended to adhere to Intel Movidius's official TraceAPI logging solution.

FLIC loggable operations fall into two categories:

- Pipeline operations.
- Message operations.

6.1 Pipeline operations

Following pipeline operations can be logged:

- Adding a plugin to a pipeline (via Pipeline::Add).
- Starting a pipeline (via Pipeline::Start).
- Stopping a pipeline (via Pipeline::Stop).
- Notification that pipeline has stopped.
- Waiting for a pipeline to end (via Pipeline::Wait).
- Deleting a pipeline (via Pipeline::Delete).

User can specify how the logging is done, that is, can provide a custom implementation for:

➢ **`virtual void LogBase::Print(const char *fmt, …)`**

but messages generated from within FLIC core are hard-coded.

Examples of generated logs:

1. "Pipe [.myPipe_] ADD_PLG [.Send]" : ".myPipe_" is the FLIC pipeline name. ".Send" is the name of the plugin that just got added.
2. "Pipe [.myPipe_] MS_LINK [.Send___.out____]→[.Recv___.in_____]": for current pipeline named ".myPipe_", a master-sender link is done between sender named "out" which belongs to plugin ".Send" and receiver "in" that belongs to plugin ".Recv".
3. "Pipe [.myPipe_] START" : pipeline named ".myPipe" was just started.

NOTE: Pipeline operation logging is disabled by default. Pipeline operations can be enabled by assigning a logging object to target pipeline, e.g. "pipe.log = myLog;". Ideally, this should be done after pipeline name is set, but after Add/Link/Start actions are triggered.

6.2 Message operations

Most `Send/Receive` calls translate at lower level into `Buffer::Push/Pop`, so logging is actually executed at Buffer level.

Message operations that can be logged via IBuffLog interface are:

- Start of a push (see IbuffLog::LogPushBegin).
- Push completed (see IbuffLog::LogPushEnd).
- Start of a pop (see IbuffLog::LogPopBegin).
- Pop completed (see IbuffLog::LogPopEnd).

Users can decide what is relevant to be logged for a given message type T, by providing a concrete implementation for **`IBuffLog`** interface.

Message Logging is disabled by default, but the user can enable it by providing a custom implementation (that matches the template type of the buffer) via **`Buffer::SetLog()`** method.

6.3 Logging control

Besides providing implementations and assigning logging objects, there are a few compile-time macros users need to set in order to enable/disable FLIC logging calls globally (i.e., within current .elf):

`FLIC_PLOG`	: enable pipeline operations logging (OFF by default)
`FLIC_OP_BEGIN_LOG`	: log Begin of Send/Receive operations (OFF by default)
`FLIC_OP_END_LOG`	: log End of Send/Receive operations (ON by default)

These macros can be overridden from Makefile like:

```
CCOPT += -D'FLIC_PLOG        =1'
CCOPT += -D'FLIC_OP_BEGIN_LOG=1'
CCOPT += -D'FLIC_OP_END_LOG  =1'
```

NOTE: Printf-based logging implementations can be expensive (thousands of cc per call). Please keep logging verbosity to minimum required.

7 Return values

All point-to-point `Send/Receive` calls (`SSender::Send`, `MReceiver::Receive`, `SReceiver::Receive`) return:

- ZERO on success;
- POSIX error code on failure (i.e. typically the POSIX `errno` value that identifies the cause).

A broadcast `MSender::Send` operation (i.e. a MSender can have multiple consumers) returns ZERO for success and non-ZERO otherwise. Users interested in the return code of a Send operation can inspect the error code returned by each receiver via `MSender::errNo[]` array (number of consumers is known via `MSender::nRecs`).

Alternatively, the user can check if a certain error code was returned by any of the consumes (e.g. EINVAL) via `MSender<T>::HasErr(uint8_t errCode)` method.

In case of error, the non-ZERO value is the bit-OR-ed value of the return codes of all consumers. If the MSender has a single consumer, the `::Send` return code coincides with `MSender::errNo[0]`

For a concrete example, see `flic/tests/flic_00_smoke/leon/testA.cpp`

NOTE: All custom Sender/Receiver implementation should return POSIX error codes and assume ZERO for successful operation.

8 IO API

8.1 Master-Sender

➢ **`void MSender<T>::Link(ISReceiver<T> *r)`**

Connects current **MasterSender** object to a compatible **SlaveReceiver** object. A Sender can connect to multiple **Receivers** via multiple **Link** calls.

➢ **`int MSender<T>::Send(T *msg)`**

Master-Sender sends msg message to all receivers. Typically this means copying the message to each consumer buffer (the actual action can be customized per receiver, via buffering policy).

This call will block current thread if consumers buffers are full.

➢ **`int MSender<T>::TrySend(T *msg)`**

Same as Send, but non-blocking, returns immediately with an error code if operation cannot be performed (i.e., no space for Send on receiver side).

8.2 Slave-Sender

➢ **`void Link(MReceiver<T> *rec)`**

Connects current Sender to a compatible MasterReceiver.

➢ **`int Pull (Io *who, T *msg, IBuffCmd<T> *c)`**

This method is invoked by the Master-Receiver (which initiates the transfer). This method can only be invoked by connected Master-Receiver.

8.3 Slave-Receiver

➢ **`void Create(uint32_t nMsg, IBuffCmd<T> *beh, ISync *s)`**

Slave-Receiver buffers incoming message. **Create** method creates storage for the messages.

Params:

`nMsg`	: max number of messages to be buffered.
`beh`	: default internal buffer behavior (Queue behavior by default).
`s`	: sync object.

Other constructor flavours:

```
void Create(uint32_t nMsg, IBuffCmd<T> *beh)
void Create(uint32_t nMsg, ISync *s)
void Create(uint32_t nMsg           )
```

➢ **`int Receive  (T *msg, IBuffCmd<T> *c = NULL)`**

For **Slave-Receiver**, this method pops a message from local buffer (i.e., Slaves buffer incoming messages).

Params:

msg : ptr to return object

c : custom Receive command (used to implement custom receives)

- **`int TryReceive(T *msg, IBuffCmd<T> *c = NULL)`**

  Same as ::**Receive**, but non-blocking.

- **`int Post(Io *who, T *msg, IBuffCmd<T> *c = NULL)`**

  Post and TryPost are invoked by **MasterSender**

- **`int TryPost(Io *who, T *msg, IBuffCmd<T> *c = NULL)`**

8.4 Master-Receiver

- **`int Receive   (T *m, IBuffCmd<T> *c = NULL)`**

  For **Master-Receiver**, this method invokes a Pull on Master-Sender which decides what data to be returned.

```
int  Receive   (T *m, IBuffCmd<T> *c = NULL) {
     return snd->Pull   (this, m, c);
}
```

- **`int TryReceive   (T *m, IBuffCmd<T> *c = NULL)`**

  Same as ::**Receive**, but non-blocking.

8.5 Timed operations

Timed Send/Receive operations are provided via:

```
int ::Send    (T *msg, TimeSpec *ts, TsType tt = REL);
int ::Receive (T *msg, TimeSpec* ts, TsType tt = REL);
```

Where the timespec can be REL(relative) or ABS(absolute).

If the Send/Receive operation cannot be executed within the timeout specified, these calls return `ETIMEDOUT`.

For a concrete example see: `flic/tests/flic_00_smoke/leon/test9.cpp`

9 Pool API

PlgPool is a stock plugin that provides buffer objects from a pool of objects. The type provided during PlgPool instantiation is the descriptor attached to each frame. Each such buffer-descriptor-type should inherit PoObj class (which contain base + size basic members).

➢ **`void Create(IAllocator *a, uint32_t nFrm, uint32_t fSize)`**

Params:

a : the buffer allocator

nFrm : the number of frames|buffers|objects to be allocated

fsize : the frame|buffer|object size

Example

Assume an image frame buffer with 4 pointers: RGBD.

On top of PoObj (Pool Object) class, one can define the user-specific params:

```
class RgbdImage: public PoObj{
  public:
      void *ptrR;
      void *ptrG;
      void *ptrB;
      void *ptrD;
      int width, height;
}
```

Once a **producer plugin** receives a frame from `PlgPool<RgbdImage>` instance via a Receive call, it must complete the `RgbdImage` descriptor based on basic PoObj information and app-specific information (which is really source/producer specific).

Example (assume a plugin body):

```
PoPtr<RgbdImage> frmPtr;
if(OK ==  in.Receive(&frmPtr)) //receive a frame from pool
{   frmPtr→width = W;
    frmPtr→height = H;
    frmPtr→ptrR = frmPtr→base + 0*W*H;
    frmPtr→ptrG = frmPtr→base + 1*W*H;
    frmPtr→ptrB = frmPtr→base + 2*W*H;
    frmPtr→ptrD = frmPtr→base + 3*W*H;
      … //do something useful ...
}
```

Later, **consumer plugins** that just read the frame above should only read the RgbdImage descriptor.

NOTE: It is strongly recommended to use when possible already defined types (e.g. types/ImgFrame.h) in order to allow for integration with other existing plugins.

10 Plugin API

- **`IPlugin::IPlugin(int maxIOs = MAX_IOS_PER_PLUG)`**

  Base Plugin constructor; "maxIOs" indicates maximum number of IOs (Senders/Receiver) that can be held by current Plugin. One can override **`MAX_IOS_PER_PLUG`** default value via Makefile options, or customize per plugin basis.

- **`void * ThreadedPlugin::threadFunc(void *param)`**

  Users are supposed to implement plugins by deriving ThreadedPlugin base-class and implementing "threadFunc" virtual pure member.

  NOTE: If the user desires more than 1 thread per plugin, its the user's responsibility to create/destroy the additional threads.

  NOTE: The thread is started when the pipe that owns current plugin is started via the `Pipeline::Start` method.

- **`void * ThreadedPlugin::Start(void *param)`**

  This method is typically invoked by owner pipeline when started via Pipeline::Start call.

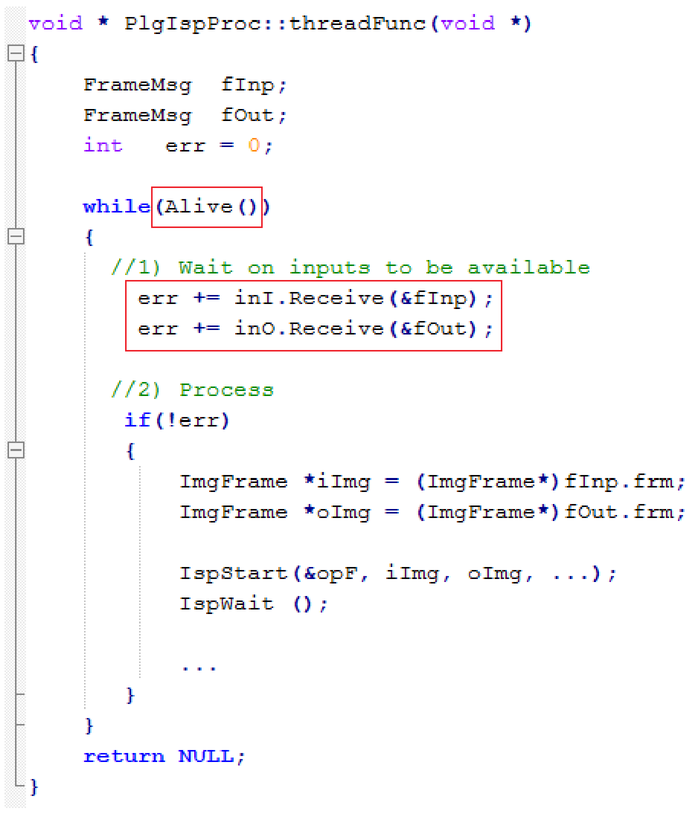
  This method starts the actual "threadFunc" above. Users can override and launch multiple threads at Start (Join would need to be updated correspondingly).

- **`void ThreadedPlugin::Stop()`**

  Plugin stop should be done in a graceful manner (i.e., not kill the current thread and leave data structures/HW setups in an undetermined state while IRQs are flying around).

  Example: assuming an ISP plugin that waits for an input and an output frame:

  - If **Stop** is invoked while ISP is currently processing the current frame (i.e., Leon "executes" work in-between `IspStart` and `IspWait` below), at the end of frame (meaning after `IspWait` returns) while-loop is evaluated and **Alive**() returns 0, the thread will exit (meaning plugin stopped).
  - If **Stop** is invoked while ISP waits for running conditions (see .Receive calls below), Plugin receivers will return error codes which will cause processing to be skipped and again, thread will exit due to **Alive**() returning 0.



```
void * PlgIspProc::threadFunc(void *)
{
    FrameMsg  fInp;
    FrameMsg  fOut;
    int    err = 0;

    while(Alive())
    {
      //1) Wait on inputs to be available
        err += inI.Receive(&fInp);
        err += inO.Receive(&fOut);

      //2) Process
       if(!err)
       {
           ImgFrame *iImg = (ImgFrame*)fInp.frm;
           ImgFrame *oImg = (ImgFrame*)fOut.frm;

           IspStart(&opF, iImg, oImg, ...);
           IspWait ();

           ...
       }
    }
    return NULL;
}
```

Figure 13: Plugin stop

NOTE: At plugin level, one could implement an early-stop mechanism and not wait till the end of current frame (i.e., above `IspWait` could return soon after Stop was invoked, and this can be done via checking periodically `Alive()` status).

➢ **`void * ThreadedPlugin::Join(void *param)`**

This method is typically invoked by owner pipeline to implement the wait for all plugins completion.

11 PIPELINE API

A pipeline is a container for multiple plugins. Grouping plugins like this simplifies Start/ Stop/ Join/ Destroy/ Delete on a group of plugins.

- **`Pipeline::Pipeline(int maxPlugs = MAX_PLUGS_PER_PIPE)`**

  Pipeline constructor. "maxPlugs" limits the number of plugins current pipe contains. One can override **`MAX_PLUGS_PER_PIPE`** default value via Makefile options, or can customize per Pipeline object.

- **`void Pipeline::Add(IPlugin *plg)`**

  Add a plugin to parent pipeline.

- **`void Pipeline::Start()`**

  Invokes `Plugin::Start()` for all plugins that belong to current pipeline. Plugs are started in the order in which they were added to the pipeline via `::Add` method.

- **`void Pipeline::Stop()`**

  `Pipeline::Stop` can be invoked from the thread that started the pipeline (e.g., POSIX_Init) or from a plugin that handles interfacing with outside world (assuming a plugin that also handles the user interface).

  This call will just invoke Stop for all plugins that belong to current pipe.

  Each Plugin also contains a reference to the Pipeline it belongs, so is able to invoke pipe termination.

12 INTER LEON COMMUNICATION

12.1 RMTO->RMTI

Inter Leon communication is achieved via 2 dedicated stock plugins: **RmtO** and **RmtI**.

One RmtO→RmtI connection is a unidirectional link and can go LOS→LRT or LRT→LOS. Multiple such links of different data type can be defined in both directions. The connection between RmtO and RmtI is a Master→Slave connection.

A blocked channel does not block other links. Responses for calls made from one Leon can come out of order from the other Leon.

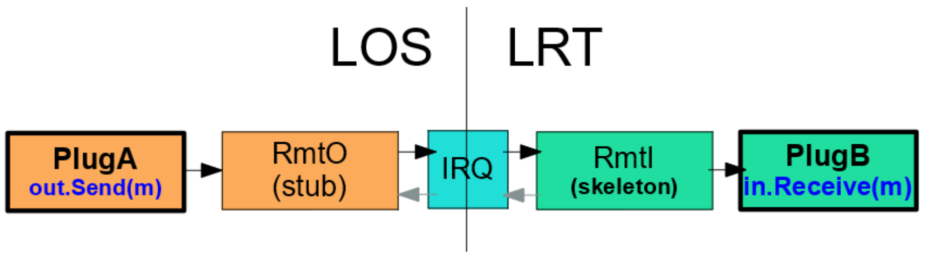


Figure 14: Inter LEON comm example

Figure 14 illustrates how {PlugA on LOS} sends data towards {PlugB on LRT} via a unidirectional data link.

The typical steps that happen internally are:

- LOS fires a REQUEST towards LRT.
- LOS thread which fired the REQUEST blocks till an ACK is received from LRT (this ACK indicates that the IRQ line is free to be used by other channels).
- LOS waits for the call RESPONSE (i.e. the REQUEST return value).
- LRT, after it processed the message eventually returns the RESPONSE back to LOS, and waits from an ACK form LOS side.

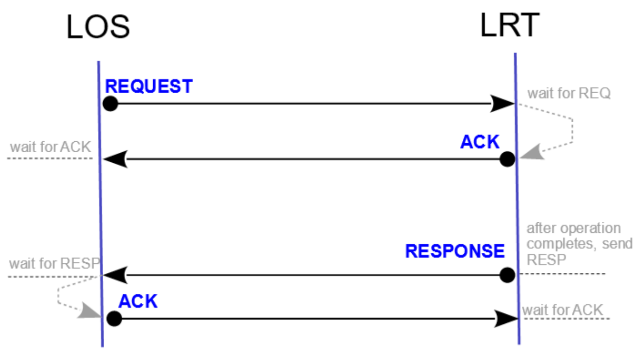


Figure 15 IRQ flow

NOTE: Even if data transfer is unidirectional, control is bi-directional: any REQUEST or RESPONSE internal calls need to be matched by an ACK from the remote side.

Stubs/Skeleton pairs get connected during construction time, e.g.:

LOS	LRT

`extern RmtI<MyMsg> skeleton; //remote on LRT`
`RmtO<MyMsg> stub; //local stub`
`...`
`stub.Create(&skeleton);`
	`RmtI<MyMsg> skeleton; //skeleton for plgB`
	`extern RmtO<MyMsg> stub; //remote stub on LOS`
	…
	`skeleton.Create(&stub);`

A **Send** from PlugA to RmtO above, will determine RmtO plugin to make a copy of the input message into non-cached address space, and forward that copy to the other Leon. In order to make sure the message reached memory, LOS (in example above) reads back the written message until **operator ==** defined for that message type returns true. This procedure requires that all messages that go cross Leon boundary to implement **operator ==**.

12.2 Makefile options

All Stubs/Skeleton plugin instances need to be added to **UNIQUE_SYMBOLS** Makefile variable to avoid symbol renaming. Example:

```
UNIQUE_SYMBOLS += stub skeleton
```

When using both Leons, user must define in Makefile following defines:

```
CCOPT     += -D'FLIC_2LEONS'
CCOPT_LRT += -D'FLIC_2LEONS'
```

12.3 Init/Destroy

At program startup, user needs to invoke **`FlicRmt::Init`**`()` which will create a few resources dedicated for FLIC inter-cpu communications (i.e. a mutex and a semaphore). When app finishes these get cleaned via **`FlicRmt::Destroy`**`()`.

12.4 Interrupts

FLIC Inter-Leon communications use 2 inter-cpu IRQ lines for all communication channels between the 2 Leons. For a given Leon:

- **1st IRQ** (FLIC_RMI_REQ_IRQ) is used notify incoming REQUESTS/RESPONSES from the other Leon.
- **2nd IRQ** (FLIC_RMI_ACK_IRQ) is used notify incomming ACKNOWLEDGEMENTS from the other Leon (i.e. every outgoing REQUEST/RESP needs to be paired with an ACK from the other side).

NOTE: For Myriad 2 these IRQ resouces can be statically set from Makefile by setting following macros: `FLIC_RMI_REQ_IRQ`, `FLIC_RMI_ACK_IRQ`.

NOTE: Myriad X drivers allow dynamic request of IRQ resources via `OsDrvDynIrqRequest`. One can enable dynamically allocated IRQs by setting `DYN_IRQ_ASSIGNMENT` macro in Makefile as: `CCOPT += -D'DYN_IRQ_ASSIGNMENT=1'`.

12.5 Passing Frame Pointers

Messages exchanged between Leons can contain PoPtr<> frame references.

Frame Pools whose frame pointers need to travel cross Leons need to be marked as SHARED at create time:

```
          pool.Create(&RgnAlloc, N_POOL_FRMS, FRM_SZ, true);
```

NOTE: For now, Shared-pool threads do not get terminated on `Pipeline::Stop` (this is work in progress).

---

13 Plugin Guidelines

13.1 Plugin Interfacing Guidelines

1. Design plugins to be re-usable. This means create plugins with common interface types so that plugins created by separate teams can be linked together easily
2. New types (that are used to template Senders/Receivers) should to be defined in an independent header under /flic/types/ directory. This header should be later included in the plugins that adopt that specific interface.
3. Maintain consistency in plugin naming, it is preferred to prefix plugin names with “plg” e.g. plgColorISP.

13.2 Plugin Development & Test Guidelines

1. The core algorithms of a plugin should be developed in isolation for debug/test, and the FLIC Plugin wrapper added as a final step. One should be able to test functionality/performance/stability of a block independent of FLIC.
2. Plugin implementation should be “light” and rely on a “common” layer, that allows forking of new plugins without duplicating a lot of code.
3. Consider grouping blocks into plugins within an application.

   For instance, if two nearby blocks can communicate via a (circular-buffer) mechanism, it makes sense to group these blocks into a single plugin (and use for example SippFramework for the underlying implementation). This will save precious DDR-BW, power and latency.
4. The user should be able to create/use multiple instances of a certain plugin type. Resource sharing of HW resources should be handled by the underlying control framework (e.g., SippFw, Fathom, drivers…). The user should ensure that the underlying control framework API should be thread-safe.
5. Ensure the plugin doesn't leak resources, for this do a Create/Destroy/Delete loop as in below example:

```
for(i=0; i<N_RUNS; i++)
{
   plg.Create(); //alloc mem and RTEMS primitives
   plg.Stop  (); //destroy RTEMS primitives
   plg.Delete(); //free mem
}
```

6. If your plugin uses additional RTEMS resources that get created in Create(…), make sure to clean them up in `::Stop`. Example:

```
void MyPlugin::Create()  {
  sem_init(&aSem, 0, 0)); //custom resource
   …
}

void MyPlugin::Stop()
{
   IPlugin::Stop();//kill all booked IOs
```

```
    //Kill additional resources
      sem_destroy(&aSem);
  }
```

7. Check **Receive**() call return values before jumping to processing.
8. If a plugin has **PoPtr** members (direct or indirect inclusion), they need to be released (e.g. via = nullptr assignment) before plugin thread(s) finish. Else destructor of a statically allocated plugin (global object) will invoke destructor of these PoPtr members which in turn will try to return to originating pool which was already destroyed (as pipeline::Stop/Delete was already invoked), thus causing some asserts to fail.

   See for example `PlgSource::releaseAllFrms()` implementation.

   This issue manifests at program termination (after POSIX_Init finishes).

13.3 Inter Plugin Communication Guidelines

1. Plugins should be able to run on either LOS or LRT (assuming no underlying framework dependency which ties a certain type of plugin to certain Leon core). For example it might be required for all SippFw ISP instances to run on same Leon core. But should be able to make that core LOS or LRT. The main aspect here is IRQ routing (some IRQs might have to be dynamically routed to one ICB at setup time).
2. Plugins should work OK independent of global RTEMS L2 settings. If data needs to be passed to remote Leon via IPC mechanism, plugin should use non-cached address space and not impose restrictions such as “this plugins will only work fine if RTEM’s L2 is disabled or set in write-through mode”.

14 Miscellaneous

14.1 IO Conditions

Typically a pugin is blocked till running input conditions are met.

Assuming for instance two inputs, “inA” and “inB”, various combinations can be achieved in the plugin thread:

- Wait for 2 mandatory inputs:
  - rc1=inA.**Receive**(&paramA);
  - rc2=inB.**Receive**(&paramB);

- Wait for first mandatory input and check second optional input:
  - rc1=inA.**Receive**(&paramA);
  - rc2=inB.**TryReceive**(&paramB);

- Wait until at least one of the first or second input is available: this approach requires that `inA` and `inB` share a sync object which is set at creation time as:
  - inA.Create(4, &sync);
  - inB.Create(4, &sync);

  Then the actual wait in plugin thread is done like:
  ```
  sync.Wait();
  ```
  Then checking of which input fired is done via inspecting return codes of TryReceive calls:
  - rc1=inA.**TryReceive**(&paramA);
  - rc2=inB.**TryReceive**(&paramB);

  (see /flic/tests/flic_03_echo_priority for a complete example)

14.2 Memory checks

FLIC can optionally override C++ **new/delete** operators, by defining `ALLOC_DEBUG` in the application Makefile, e.g.: `CCOPT += -D'ALLOC_DEBUG'`. This allows basic memory leaks, and memory overflow checks to be performed.

All pending allocs are logged in a buffer of size `ALLOC_MAX`, which can be overridden from Makefile.

Example: `CCOPT += -D'ALLOC_MAX=256'`

All allocs/deletes can be printed via: `CCOPT += -D'ALLOC_PRINT=1'` (by default, prints are disabled).

All allocs return locations aligned to NEW_ALIGN value (default = 16, user can override via Makefile).

See `\mdk\testApps\components\flic\RTEMS_alu_08_new_align`

NOTE:	gcc's dynamic allocation (new operator) does NOT return “properly” aligned storage for structures that have in their definition alignment attributes.

All allocs also reserve some extra bytes at the end of the buffer where a canary value is placed. One can check canaries status via **`CheckOverflow`**`();`

All current allocs could be displayed via **`ShowAllocs`**`()`. Allocs that happen in source files that include directly or indirectly `Flic.h`, will also benefit of `__FILE__`, `__LINE__` identification (pointing where allocs happen, see `DEBUG_NEW` macro). Remaining allocations will be mapped into `__FILE__` = "N/A", `__LINE__` = 0. See *`flic_00_alloc_global`* test for a concrete example.

At the end of app, one could check for memory leaks via **`CheckMemLeaks`**`()`. Basically all allocs still left in the log allocation table are considered leaks.

The user could disregard default FLIC memory checker and provide their own implementation.

14.3 Interrupts

WARNING: High load/frequent IRQs (that is, an IRQ every few thousands clock-cycles) should be serviced in the IRQ handler and not deferred to a waiting thread (which would get notified via an event for example) as that would cause +1 thread switch for each IRQ and would kill CPU performance (see `RTEMS_alu_23_irq_inline`).

Real example: a timer fires IRQ at each 2400 cc and IRQ handler is ~ 500 x NOPs.

- if IRQ is handled directly => CPU load = **33.85%**
- if IRQ is deferred to a waiting thread => CPU load = **99.70%**

NOTE: On the other hand, frame-level expensive IRQ handlers (e.g., Video Encode EOF for H.265 could take thousands of CCs to execute) and should be handled in a waiting thread.

14.4 Template code size per type

Each new message/frame type added to the system will increase the code size by ~1.2KB.

As plugins are meant to be re-used in multiple pipelines and the outputs of one plugin are the inputs to another it is recommended to try and reuse already implemented types as much as possible (e.g., ImgFrame).

14.5 Dynamic allocation

It's recommended to do dynamic allocation only at setup stage, not during runtime.

NOTE: When allocating a derived object via new, but deleting via a pointer to base class, make sure the base-class contains a virtual destructor (even if it's empty), to avoid undefined behavior. See: http://stackoverflow.com/questions/25220229/in-c-inheritance-derived-class-destructor-not-called-when-pointer-object-to-b.

Undefined behavior scenario output on Leon:

```
UART: new  0x701bd598
UART: Program heap: free of bad pointer 701BD598 -- range 701BD18E -
701CDE00
```

Other observed undefined behavior side-effects:

- Leon hang in **new** call
- Leon hang in **sem_destroy** call
- …

14.6 RTEMS thread policies/suspicious hangs

Under RTEMS, Posix default thread policy is SCHED_FIFO (meaning: no timeslicing, the task needs to yield CPU explicitly). So new threads that inherit **POSIX_Init** attributes or use default values might lock CPU for long times and cause undesired frame drops/hangs.

It is recommented to set threading policy to SCHED_RR to allow fair CPU utilization among plugins.

Sample code:

```
...
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED));
pthread_attr_setschedpolicy (&attr, SCHED_RR));
...
```

NOTE: All FLIC Plugin threads use by default SCHED_RR policy.

14.7 BUFFERING POLICIES

By default, Receiver objects do the message buffering, and the policy used is Queue.

Users can define custom buffering policies to meet application-specific requirements and this is typically done by:

- Defining a class that implements the custom logic.
- Setting a default executor or providing customized arguments to Receive/Send(tbd).

See `flic_01_buffer_custom` for a concrete example.

14.7.1 Queue

Queue buffering means messages are popped in the same order they were pushed/posted.

14.7.2 Overwrite oldest

This behavior allows a buffer to just keep most recent N messages (where N = buffer size), and automatically discard older messages.

Example: Assume we do 6 pushes of messages (A,B,C,D,E,F) in a buffer of 4 entries.

- if buffer uses **Queue** policy, at 5th push producer would block and buffer would contain: A,B,C,D.
- if buffer uses **OvrerwriteOldest**, all 6 pushes would succeed and buffer would contain C,D,E,F which are the most recent messages.

14.7.3 Custom buffering example

Users can customize Push/Pop operations to meet specific app requirements.
Naive custom/app examples:

- want to **Receive** the message that contains a frame with time-stamp closes to a certain value.
- want to **Receive** the message that contains largest ROI (region-of-interest).
- want to **Receive** a message that contains just VIDEO frame (and discard anything else).
- ...

See `flic_01_buffer_custom` for a concrete example.

14.8 Synchronized outputs

Assuming a plugin produces multiple different outputs in an inherently synchronized manner (e.g. Frame + associated Metadata). Some consumer plugs would only care about one output (e.g. Frame that goes to LCD), but some plugs might need the synchronized pair of Frame + Meta for encoding.

In this case it's preferable for the producer plugin to expose both:

- individual outputs:
```
MSender<PoPtr<ImgFrame>> outFrame;
MSender<PoPtr<Metadata>> outMeta;
```
- synchronized outputs:
```
MSender<SyncedMsg> outSynced;
```
  where:
```
typedef struct{
   PoPtr<ImgFrame> oFrame;
          PoPtr<Metadata> oMeta;
}SyncedMsg;
```

This would avoid the need of an additional timestamped-based synchronization block.

14.9 Adapter

Sometimes IO types plugins aren't compatible and some adaptation is required in order to connect two plugins together, e.g.:

```
plgA.out.Link(&plgB.in); //=> compile error due to type mismatch
```

For MSender→SReceiver link, `SRecAdapt` threadless helper plugin is provided to ease the task. This adapter plugin has an input of type **`TIn`** and an output **`TOut`** which and has to be linked in between the incompatible IOs:

```
plgA.out.Link(&plgX.in);
plgX.out.Link(&plgB.in); //=> compile OK
```

User has to implement below method, which is invoked when `plgA.out.Send(TI *msg)` is executed

➢ **`void Convert(const TI *dIn, TO *dOut) = 0;`**

For a concrete example see `flic/tests/flic_12_plg_adapter`

NOTE: Some cases cannot be properly solved by using Adapter approach (e.g. assume a plugin pushes a plain-pointer whereas consumer plugin expects a PoPtr (shared-pointer) managed by a proper Pool).

14.10 Plugin-groups

When a certain functionality can only be exposed via a complex set of plugins & connections, such functionality could be organized as a reusable plugin-group. This way other users can just import & use the plugin-group (which is basically a sub-pipeline) in a simple manner.

Once defined, a Plugin-group can be added to an existing pipeline via `PlgGroup::AddTo` method.

Public plugin-groups should be placed in `flic/pipelines/` folder.

For a concrete example see `flic/tests/flic_11_plg_group`.

NOTE: The concept of Plugin-group (see PlgGroup interface) is basic: once added to a pipeline, user should not attempt stop/start/destroy/delete of that group independently of the pipe it was added to. Plugin-group sole purpose it to ease the creation of pipelines that require standard constructs (e.g. ISP).

14.11 Atomic section

To perform a set of operations in an atomic fashion the Atomic objects could be used and just use Enter/Leave methods to guard the critical section. In `Atomic::Enter` and `Atomic::Leave`, IRQs and preemption are disabled.

Atomic sections can be nested.

15 Tests

Tests can be found in `mdk/common/components/flic/tests`.
One can run `./runRegresion.sh` (in /tests folder) to see the running status of all self-checking tests.
Below is a list of existing tests/learning examples.

Test name	Description
flic_00	Running multiple independent plugins.
flic_00_alloc_buff	Buffer Alloc/Destroy cycle.
flic_00_alloc_checks	• override new/delete • check constructor/destructor calls • check mem leak
flic_00_alloc_global	Print all allocs.
flic_00_alloc_plug	Plugin with custom resources Alloc/Destroy cycle.
flic_00_alloc_pool	Pool alloc/free cycle via Region/Heap allocators.
flic_00_frame_fwd	Frame propagation from Source to End-consumer.
flic_00_hello_00	Hello World : pass 4 messages from Sender to Receiver (same thread).
flic_00_hello_01	Hello World : pass 4 messages from Sender to Receiver (different plugs).
flic_00_hello_02	FramePool Receive + Forward.
flic_00_hello_03	Typical plugin with input and output (see plgB).
flic_00_id_map	Print ID strings.
flic_00_log	Log Send/Receive calls.
flic_00_posix_receiver	A Slave-Receiver implementation based on POSIX's message-queue.
flic_00_queue	Queue test: Push, Pop, TryPush, TryPop.
flic_00_prc	Two Leons: Internal basic RMI test.
flic_00_prc_a	Two Leons: synchronization-barrier test.
flic_00_prc_b	Two Leons: passing POD messages from LOS to LRT (multiple chains).
flic_00_prc_c	Two Leons: passing POD messages cross Leons, messages cross the inter-Leon boundary multiple times via Stub/Skeleton plugins.
flic_00_prc_d	Two Leons: same as flic_00_prc_c, but without buffer plugins, a Skeleton connects directly to a Stub.
flic_00_prc_e	Two Leons: passing messages that contain frame references (PoPtr).
flic_00_prc_f	Two Leons: passing messages that contain frame reference + return to original Leon.
flic_00_smoke	Smoke test; touching most core features.
flic_00_stress_buffer	**Stress test**: single Buffer written by 64 x Writes and read by 64 x Readers.

Test name	Description
flic_00_thread_priority	Setting thread priority @ start.
flic_01_buffer_custom	User custom `::Receive()` example.
flic_01_echo	Echo test ("infinite" loop).
flic_01_echo_sys_nfo	Same as 01_echo, with `cpu_usage` dummy `printf`
flic_01_echo_template	Same as 01_echo, but plugs-templated.
flic_01_get_closest_ts	"Get closest TS" `::Receive` example.
flic_01_stress_hist	**Stress test**: a 128 element histogram is send by 128 senders to a unique receiver.
flic_01_stress_ovr	**Stress test**: 32 sender plugs push data towards a single Receiver that uses OvrOldest policy.
flic_02_multi_pipe	Multiple pipes test.
flic_03_echo_priority	Input grouping (`::TryReceive` on inputs).
flic_04_tnf	TNF(Temporal Noise Filtering) example.
flic_05_super_frame	Super-Frame example : split a frame in parts, give to consumers, wait for results, push resulting super-frame on output.
flic_07_destroy	App Start/Stop cycle.
flic_08_roi	Multiple ROI per frame detect & forward.
flic_09_cnn_a	Dummy sequential CNN Age + Gender pipe.
flic_09_cnn_a_tmpl	Dummy sequential CNN Age + Gender pipe (using templates).
flic_10_sync_frames	Plugin receive 2 frames and propagate on output a single message containing both frame refs.
flic_11_plg_group	Plugin group example.
flic_12_plg_adapter	Match incompatible IOs via helper Adapter plugin.