Interprocess Communication (IPC)

Description

Courier IPC provides a mechanism to send Courier messages to other applications (aka. processes) running on the same processor. Communication between physically distinct processors is not supported.

This tutorial describes how you may enable IPC for your applications. We assume, that you are familiar with the basic Courier concepts like messages, components, Courier XCDL and generator, etc.

Describe IPC System Structure

Enabling IPC in Courier

IPC is not enabled per default. To enable IPC the feature `FEATSTD_IPC_ENABLED` has to be turned on in the CMake file. This can be achieved either in the CMake GUI, on the command line or by adding

```
# ------------------------------------------------------------------------
# Enable Courier IPC feature
set(FEATSTD_IPC_ENABLED ON)
```

to the `CMakeLists.txt` file of all applications that need IPC.

Define the Connections

The next step to enable IPC is to define which applications are communicating with each other and which messages they are using for communication. In other words, you have to define which applications take part in the IPC by assigning process IDs, define the connections between those applications and also define the messages which shall be exchanged. If two applications want to exchange messages, the messages have to be known by both of them.

Defining the IPC communication structure is done as part of the AppCDL.xcdl. One thing to consider is, that this information has to be shared between all applications that are part of the IPC. So it is probably a good idea to put all the IPC related configuration into a dedicated file which can be accessed by all applications resp. their development teams.

A simple example for such an IPC configuration could look like this:

```
<definition
    xsi:noNamespaceSchemaLocation="../../../../cgi_studio_courier/schema/Courier.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <generator generate="false"/>
    <ipc>
        <process name="Harold" uid="" />
        <process name="Maude" uid="" />

        <connection process1="Harold" process2="Maude" />
    </ipc>
</definition>
```

In our example we have two applications named HaroldApp and MaudeApp that exchange messages over IPC.

- First two processes for the applications are defined. Behind the scenes each process is assigned an unique ID, but within the CDL the ID can be referenced by the process' name.
- Second, we configure a connection between those processes.

A connection is always bidirectional so there is no need to define a separate connection the other way round. However *process1* takes a special role when it comes down to actually opening the connection. Because of the way IPC is handled on the OS level, one process has to create the connection on OS level before the second process can open it. The first process is called the *creator* in our context.

Describe Messages Used in IPC

We can describe the messages used for IPC using the XCDL (Courier Definition Language), an XML dialect used throughout Courier to define messages. As the messages are sent back and forth between two applications, it is recommended, that the message definition file is shared between the applications. It is possible to have a separate file for each connection including only the messages exchanged by the two applications on that channel. This should help to keep dependencies at a minimum.

```
    <message name="ToMaudeControllerAndLocalControllerSeqIpcMsg" serializable="true"
distribution="sequential" uid="{d7cfaf4b-219f-456b-843c-9a735aab7880}" >
      <subscribers>
          <subscriber id="Controller" process="Maude" />
          <subscriber id="Controller" />
      </subscribers>
    </message>
```

Defining messages for IPC is basically the same as for non-IPC Courier based applications. The only addition is, that, if the message should be distributed to a component in another application, the process ID of the concerned application has to be added. Of course, a message can be delivered locally and over IPC at the same time. In our example the message is sent to the local controller and to the controller of Maude.

Build AppCDL

Once we have defined all the messages and connections we have to include them in the involved applications. Usually, an application has a root XCDL file, where also local messages are defined or included from other CDL files.

```
<definition
    xsi:noNamespaceSchemaLocation="../../../../cgi_studio_courier/schema/Courier.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">



    <include generate="false" addIncludeToCode="false"
href="../Shared/IpcHaroldAndMaudeConnections.xcdl" />
    <include generate="true" href="HaroldAppMsgs.xcmdl" />
    <include generate="true" href="../Shared/HaroldAndMaudeIpcMsgs.xcmdl" />
    <include generate="true" href="AppPlatform/MessageFactoryConfiguration.xcmdl" />



    <!-- Generate C++ file using
            CMake: -o ${COURIER_XCMDL_OUTPUT_DIR} (.../cgi_studio_courier/src)
            CL:    -o ../../../ (1st ../ for XCMDL) -->
    <generator generate="true" location="CourierIpcHaroldApp">
        <header>
            <prolog file="../../../../cgi_studio_courier/CourierCopyright.txt" />
        </header>
        <source>
            <prolog file="../../../../cgi_studio_courier/CourierCopyright.txt" />
        </source>
    </generator>

    <ipc processName="Harold" />


</definition>
```

We include the XCDL files for the messages used in Harold internally, the IPC connections, the IPC messages and have to announce that the application *Harold* has the process id Harold.

Generating IPC Information

Once all things are put together, the Courier generator has to be used to generate all the source files for all the messages and the IPC information. The generator produces almost all information needed for successful IPC.

Process IDs

The generated file `AppCDL.h` contains the constant definitions for the process IDs of Harold and Maude.

```
const ::Courier::IpcProcessId HaroldProcessId = 1;
const ::Courier::IpcProcessId MaudeProcessId = 2;
```

Connection Info

The second product of the generator is the connection table containing an entry for each connection (in our example one entry).

```
    static const ::Courier::IpcConnectionInfo gTable[] = {
        ::Courier::IpcConnectionInfo
(true, ::MaudeProcessId, ::AppPlatform::AppEnvironment::GetChannelConfiguration(0)),
    };
```

This table is generated static, so we need a way to give the Courier::IpcComponent access to it. That is why the following function is generated:

```
Courier::IpcConnectionInfoAccessorBase* GetIpcConnectionInfoAccessor();
```

Message Creation

To create "empty" messages which are filled with the deserialized data coming via IPC Courier a message "factory" (instantiation) is generated:

```
class HaroldIpcMessageCreation : public ::Courier::IpcMessageCreation
{
    public:
        virtual ::Courier::Message * CreateMessage(::Courier::UInt32 msgId) const;
};

::Courier::Message * HaroldIpcMessageCreation::CreateMessage(::Courier::UInt32 msgId) const
{
    ::Courier::Message * msg = 0;
    switch (msgId) {

        ...
        case ::ToHaroldControllerAndLocalControllerSeqIpcMsg::ID:
            msg =
```

```
COURIER_MESSAGE_NEW(::ToHaroldControllerAndLocalControllerSeqIpcMsg)();
            break;
        ...

        default:
            break;
    }
    return msg;
}
```

IPC Messages

Message definitions for the IPC messages are also created. In our example they are generated into the files `HaroldAndMaudeIpcMsgs.h` and `HaroldAndMaudeIpcMsgs.cpp`. It is also possible to generate them independently so that the applications don't need to share source files but XCDL files only.

Integrating IPC into the Application

Now we will see what needs to be done to get an application ready for IPC.

Platform Dependent Channel Configuration

For each defined connection an instance of FeatStd::Internal::IpcChannel is used to perform the actual data transmission back and forth. The implementation of the channel is dependent on the used platform. The needed configuration parameters varies from platform to platform. On an abstract point of view, a channel can be seen as two queues, one for sending messages and one for receiving. Each queue has a maximum number of entries, each entry has a maximum size and each entry needs memory.

Consequently the set of configuration parameters for all platforms are:

- max. outgoing message buffers
- outgoing message buffer size
- max. incoming message buffers
- incoming message buffer size
- pointer to message buffers

Microsoft Windows Platform

The simplest platform in terms of IPC channel configuration is Win32. Here the OS completely takes care of all buffer handling. The only parameters the application has to define is the total size of memory to be used for incoming and the total size of memory used for outgoing messages. The message buffer counts are simply "one", the pointers to the message buffers are set to zero.

To make things easier for the application, FeatStd provides the FeatStd::Win32::StaticWindowsChannelBufferConfig template that can be used to create a proper

configuration object.

```
    static FeatStd::Win32::StaticWindowsChannelBufferConfig<2048, 2048> gChannelBufferConfig;

    // ------------------------------------------------------------------------
    FeatStd::IpcChannelConfiguration * AppEnvironment::GetChannelConfiguration(Courier::UInt32
/* idx */) {
        /* we want all channels have the same config, we don't need to allocate memory ergo we
can use the same object for all channels. */
        return &gChannelBufferConfig;
    }
```

GHS Integrity Platform

On Integrity, the communication is done via Integrity connections. The IPC channel has to hold a set of receiving buffers for incoming transfers. Applications are responsible for allocating the buffers and has control over buffer allocation. The interface for this is FeatStd::IpcChannelConfiguration, which has to be implemented by the application.

For each connection one FeatStd::IpcChannelConfiguration instance is needed and has to be made accessible via GetChannelConfiguration() method inside your AppPlatform implementation. The generated IPC information relies on the existence of this interface. For static buffer allocation a template can be used to create a table of static buffers.

```
    static FeatStd::Integrity::StaticIntegrityChannelBufferConfiguration<10, 500>
gChannelBufferConfig[HAROLD_IPC_CONNECTION_COUNT];

    // ------------------------------------------------------------------------
    FeatStd::IpcChannelConfiguration * AppEnvironment::GetChannelConfiguration(Courier::UInt32
idx)
    {
        FEATSTD_DEBUG_ASSERT(idx < HAROLD_IPC_CONNECTION_COUNT);
        if (idx < HAROLD_IPC_CONNECTION_COUNT) {
            return &gChannelBufferConfig[idx];
        }
        return 0;
    }
```

Posix Platform

For all Posix based platforms (mainly Linux), the IPC channel is implemented using Posix message queues. In this case the buffer handling is done by the OS. The needed configuration consists of:

- number of outgoing message buffers
- outgoing message buffer size
- number incoming message buffers
- incoming message buffer size

```
    static FeatStd::Posix::StaticPosixChannelBufferConfig<10, 2048, 10, 2048>
gChannelBufferConfig;

    FeatStd::IpcChannelConfiguration * AppEnvironment::GetChannelConfiguration(Courier::UInt32
/* idx */) {
        /* we want all channels have the same config, we don't need to allocate memory ergo we
can use the same object for all channels. */
        return &gChannelBufferConfig;
    }
```

As no buffers are needed, it is possible to use a single configuration object for all channels, if all channels shall have the same configuration. Also for Posix a template exists for easy use.

Enable IPC in the Application

In order for IPC to run, the application has to instantiate and activate a set of objects which are described here. Here is an example of the Harold application class:

```
class HaroldApp
{
    public:
        HaroldApp();
        virtual ~HaroldApp();

        bool Init(AppPlatform::AppEnvironment * appEnvironment);
        bool Run();
        bool Finalize();

    private:
        AppPlatform::AppEnvironment         * mAppEnvironment;

        // The message receiver
        Courier::ComponentMessageReceiver   mMsgReceiver;
```

```
        // The application specific ControllerComponent
        HaroldAppControllerComponent        mControllerComponent;

        // Platform independent external communication component
        Courier::ExtCommComponent           mExtCommComponent;

        // Platform independent external output handler
        ExtCommOutputHandler                mExtCommOutputHandler;

        //
        HaroldIpcConnectionControlPolicy    mIpcPolicy;
        //
        HaroldIpcComponent                  mIpcComponent;

        Courier::IpcStaticBuffer<2048>      mSendReceiveBuffer;
};
```

IpcComponent

The most important component for IPC is the Courier::IpcComponent. The application has to keep an instance of Courier::IpcComponent and attach it to a message receiver. First the IPC component has to be initialized:

```
    bool retVal = mIpcComponent.Init(HaroldProcessId, &gMessageCreation,
GetIpcConnectionInfoAccessor(), &mSendReceiveBuffer, &mIpcPolicy);
```

Courier::IpcComponent needs then following data for proper operation.

Local Process ID

The local process ID has been defined in the application's CDL file (see Build AppCDL).

Courier::IpcComponent has to know which process ID it shall use.

Message Instantiation

Incoming data is deserialized inside Courier::IpcComponent. Consequently, Courier::IpcComponent has to create empty message instances. For this, a message "factory" is generated as described in Generating IPC Information . This message instance is then filled with content (deserialization) and posted as usual.

Connection Information

Obviously Courier::IpcComponent also needs the information about connections used by the application. This information is passed via an instance of Courier::IpcConnectionInfoAccessorBase. For each application a derived implementation of this class is generated, which can be accessed by the (also generated) function GetIpcConnectionInfoAccessor().

De-/Serialization Buffer

For serializing and deserializing messages Courier::IpcComponent needs a buffer. The application has full control over how this buffer is allocated. It has to pass the buffer to Courier::IpcComponent during initialization.

The IpcPolicy

During operation connections to remote peers are opened, closed and re-opened. By implementing its own version of Courier::IpcPolicy the application can control when Courier::IpcComponent is allowed to open a connection. If a Courier::IpcPolicy is provided, Courier::IpcComponent asks the policy instance for authorization before opening a connection.

```
class MaudeIpcConnectionControlPolicy : public Courier::IpcPolicy {

public:
    static const Courier::UInt32 cTimeOut = 3000;

    MaudeIpcConnectionControlPolicy():
        mTimeout(Courier::Platform::TicksType::Infinite)
    {}

    virtual void OnConnectionEstablished(Courier::IpcProcessId /* peerPid */) {
    }

    virtual void OnConnectionLost(Courier::IpcProcessId /* peerPid */) {
        mTimeout = Courier::Platform::Ticks(Courier::Platform::TicksType::Now);
    }

    virtual bool CanConnect(Courier::IpcProcessId /* peerPid */) {
        bool rc = (mTimeout.IsInfinite() || (mTimeout.GetExpiredTime() > cTimeOut));

        if (rc) {
            // store timestamp of last time connect was authorized
```

```
            mTimeout = Courier::Platform::Ticks(Courier::Platform::TicksType::Now);
        }
        return rc;
    }

private:
    Courier::Platform::Ticks mTimeout;
};
```

This example sketches how to implement an IpcPolicy that allows a reconnect after a certain time has passed since the last connection loss. The idea is, to slow down reconnects a bit to reduce system load. Of course this example misses handling each connection separately. Providing an Courier::IpcPolicy is optional, but it may help to better control IPC based on facts that only the application knows.

Activating The IpcComponent

As any other component that shall be executed and able to receive messages the Courier::IpcComponent has to be attached to a Courier::ComponentMessageReceiver.

```
    // Attach IPC component -> eventually maybe done in Courier
    mMsgReceiver.Attach(&mIpcComponent);
    // Before sending the start up (resp. the first) message be sure to activate all message
receivers
    mMsgReceiver.Activate();
```

After that the application is able to send and receive messages via IPC.

IPC Sample Applications

There are several sample applications shipped so that you may have a look how things might be implemented in your own application.

First of all there are HaroldApp and MaudeApp which can be found in the following directories:

- cgi_studio_courier_apps/src/IpcApplications/CourierIpcHaroldApp
- cgi_studio_courier_apps/src/IpcApplications/CourierIpcMaudeApp

They are sending messages from one to the other and receive responses. Most of the code snippets in this tutorial are taken from HaroldApp.

Second SampleApp got a new partner application, called RemoteApp, which may be used do send key input ('1' to '9') via IPC messages and let you therefore control the speed value of SampleApp. Main purpose is to demonstrate how easily SampleApp (which is using Visualization and

DataBinding) might be used with IPC feature. When building the feature has to be explicitly enabled in CMAKE.

Handling IPC Component Errors

The Courier::IpcComponent base class has no default error handling implemented but provides a virtual method which may be overwritten. The following code snippet shows how this can be implemented:

```
class HaroldIpcComponent : public Courier::IpcComponent
{
    protected:
        virtual void OnError(Courier::IpcErrorCode::Enum errorCode, Courier::IpcConnection *
connection, FeatStd::UInt32 msgId, const Courier::Message * msg)
        {
            // Always check connection and msg pointer if they are valid
            FEATSTD_UNUSED3(connection,msgId,msg);
            switch(errorCode) {
                case Courier::IpcErrorCode::ReceiveFailed: {
                    printf("\nReceiving of an IPC message failed\n");
                    break;
                }
                case Courier::IpcErrorCode::SendFailed: {
                    if(msg!=0) {
                        printf("\nSending of IPC message '%s' failed \n",msg->GetName());
                    }
                    break;
                }
                default: {
                    // another error occured.
                }
            }
        }
};
```

Revision #8
Created 2023-02-27 05:20:42 UTC by Tsuyoshi.Kato
Updated 2025-06-01 22:19:56 UTC by Elisabeth Zauner