Platform Configuration

Description

Some platform specific topics are described here.

Platform Definition

A Courier platform definition describes the complete setup for a dedicated platform. This comprises the implementation of the various platform interfaces defined in Courier::Platform but also stores the Candera configuration and the characteristics of the target build environment (like OS, Compiler, CPU, etc.).

Courier pre-defines a few sets of platform definitions, which are located in the folder

```
./cgi_studio_courier/src/CourierPlatformDefs/<PlatformName>/
```

and Courier may use one of them, which is selected via CMake. See Platform Configuration & Build Process.

If there is a need to adapt a platform definition, like different Candera setup, additional platform configurations, compiler settings, etc. simply duplicate and modify or create a completely new platform definition folder, which has a cmake file with the name *<AnyPlatformName>Def.cmake*. If the new platform definition doesn't reside in the CourierPlatformDefs folder a simple Courier CMake commando (COURIER_EXTENDED_SEARCH_PATHS), applied accordingly to your Courier application project, will help to find it.

Platform Definition File

The platform definition file holds the complete setup for the platform, as the example Courier's Platform shows:

Config macros

Config macros are used to source re-useable configuration out of any concrete platform definition. Currently they are used to define compiler specific sets for the platform definitions, e.g. for MSVC, gcc, etc. Generally these config macros are included at the beginning of the platform definition file.

Take a look at the config macros for MSVC setup:

```
CourierConfigMacro(MSVCBase "Base compiler flag set for Microsoft Visual Studio")

    CourierAddConfigMacroRequirement(COURIER_TOOLCHAIN_ID STREQUAL "MSVC")

    if(FEATSTD_64BIT_PLATFORM)
        SET(tmp_linker_flags "/STACK:10000000 /machine:X64 ")
    else()
        SET(tmp_linker_flags "/STACK:10000000 /machine:X86 ")
    endif()

    CourierSetConfigMacroProperties(
        # C++ compiler
        CMAKE_CXX_FLAGS                             "/D_VARIADIC_MAX=10 /DWIN32 /D_WINDOWS
/DWINVER=0x0601 /D_WIN32_WINNT=0x0601 /Zm1000 /GR /MP"
        CMAKE_CXX_FLAGS_DEBUG                       "/D_DEBUG /MDd /Ob0 /Od /Z7 /RTC1
/DFEATSTD_BUILD_MODE=FEATSTD_BUILD_MODE_DEBUG"
        CMAKE_CXX_FLAGS_MINSIZEREL                  "/MD /O1 /Ob1 /DNDEBUG
/DFEATSTD_BUILD_MODE=FEATSTD_BUILD_MODE_MINSIZEREL"
        CMAKE_CXX_FLAGS_RELEASE                     "/MD /O2 /Ob2 /DNDEBUG
/DFEATSTD_BUILD_MODE=FEATSTD_BUILD_MODE_RELEASE"
        CMAKE_CXX_FLAGS_RELWITHDEBINFO              "/MD /O2 /Ob2 /Z7 /DNDEBUG
/DFEATSTD_BUILD_MODE=FEATSTD_BUILD_MODE_RELWITHDEBINFO"
        CMAKE_CXX_STANDARD_LIBRARIES                "kernel32.lib user32.lib gdi32.lib
winspool.lib shell32.lib ole32.lib oleaut32.lib uuid.lib comdlg32.lib advapi32.lib"

        #    C compiler
        CMAKE_C_FLAGS                               "/D_VARIADIC_MAX=10 /DWIN32 /D_WINDOWS
/DWINVER=0x0601 /D_WIN32_WINNT=0x0601 /W4 /Zm1000 /MP"
        CMAKE_C_FLAGS_DEBUG                         "/D_DEBUG /MDd /Ob0 /Od /Z7 /RTC1
/DFEATSTD_BUILD_MODE=FEATSTD_BUILD_MODE_DEBUG"
        CMAKE_C_FLAGS_MINSIZEREL                    "/MD /O1 /Ob1 /DNDEBUG
/DFEATSTD_BUILD_MODE=FEATSTD_BUILD_MODE_MINSIZEREL"
        CMAKE_C_FLAGS_RELEASE                       "/MD /O2 /Ob2 /DNDEBUG
/DFEATSTD_BUILD_MODE=FEATSTD_BUILD_MODE_RELEASE"
        CMAKE_C_FLAGS_RELWITHDEBINFO                "/MD /O2 /Ob2 /Z7 /DNDEBUG
```

```
/DFEATSTD_BUILD_MODE=FEATSTD_BUILD_MODE_RELWITHDEBINFO"
        CMAKE_C_STANDARD_LIBRARIES                  "kernel32.lib user32.lib gdi32.lib
winspool.lib shell32.lib ole32.lib oleaut32.lib uuid.lib comdlg32.lib advapi32.lib "

        # linker EXE
        CMAKE_EXE_LINKER_FLAGS                      ${tmp_linker_flags}
        CMAKE_EXE_LINKER_FLAGS_DEBUG                "/INCREMENTAL /debug"
        CMAKE_EXE_LINKER_FLAGS_MINSIZEREL           "/INCREMENTAL:NO"
        CMAKE_EXE_LINKER_FLAGS_RELEASE              "/INCREMENTAL:NO"
        CMAKE_EXE_LINKER_FLAGS_RELWITHDEBINFO       "/INCREMENTAL /debug"

        # linker Modules
        CMAKE_MODULE_LINKER_FLAGS                   ${tmp_linker_flags}
        CMAKE_MODULE_LINKER_FLAGS_DEBUG             "/debug /INCREMENTAL"
        CMAKE_MODULE_LINKER_FLAGS_MINSIZEREL        "/INCREMENTAL:NO"
        CMAKE_MODULE_LINKER_FLAGS_RELEASE           "/INCREMENTAL:NO"
        CMAKE_MODULE_LINKER_FLAGS_RELWITHDEBINFO    "/INCREMENTAL:NO /debug"

        # linker DLLs
        CMAKE_SHARED_LINKER_FLAGS                   ${tmp_linker_flags}
        CMAKE_SHARED_LINKER_FLAGS_DEBUG             "/debug /INCREMENTAL"
        CMAKE_SHARED_LINKER_FLAGS_MINSIZEREL        "/INCREMENTAL:NO"
        CMAKE_SHARED_LINKER_FLAGS_RELEASE           "/INCREMENTAL:NO"
        CMAKE_SHARED_LINKER_FLAGS_RELWITHDEBINFO    "/INCREMENTAL:NO /debug"

        # resource compiler
        CMAKE_RC_FLAGS    " "

        # tool chain features
        COURIER_DEPRECATED                          "__declspec(deprecated)"
        COURIER_DEPRECATED_MSG                      "__declspec(deprecated(msg))"
    )
CourierConfigMacroEnd()

CourierConfigMacro(MSVC_Candera "MSVC flags with warning level 3 and enabled exception
handling" PARENT MSVCBase)
    CourierSetConfigMacroProperty(CMAKE_CXX_FLAGS "/W4 /EHsc" ADD)
CourierConfigMacroEnd()

CourierConfigMacro(MSVC "Default compiler configuration for Microsoft Visual Studio" PARENT
```

```
MSVC_Candera)
CourierConfigMacroEnd()
```

General Platform setup

The basic platform setup begins with the macro *CourierPlatformDef* setting the platform name and a short description about this platform. Various settings are included in the platform definition block, like the choices of toolchains, various platform configurations (e.g. for Host, Target, etc.), and general properties (valid for all platform configurations).

Platform configuration

A platform definition is additionally separated one or more sections of platform configurations. Usually a host (e.g. for simulation on Win32) and a target configuration are defined. Though they are typically named *Host* and *Target*, the names of these platform configurations are changeable.

Platform Implementation

Besides the platform definition file, also the implementation of the Courier::Platform interfaces is essential (see also OS Abstraction), which can be found in the subfolders of the various platform definition. For each platform configuration which is defined in the platform definition file, a subfolder with the name of the platform configuration has to exist.

E.g. if there are the several platform configurations, it has to look like this:

```
<AnyPlatformDefsFolder>/<PlatformName>/Integrity_ARM
<AnyPlatformDefsFolder>/<PlatformName>/Linux_ARM
<AnyPlatformDefsFolder>/<PlatformName>/Simulation_Win32_x64
<AnyPlatformDefsFolder>/<PlatformName>/Simulation_Win32_x86
<AnyPlatformDefsFolder>/<PlatformName>Def.cmake
```

In this platform configuration folders, the realizations of the Courier::Platform interfaces are located. There are at least two possibilities how this can be done:

1. Directly code in the platform configuration folder the implementation of the Courier::Platform interface of choice and name the corresponding source and header file like Courier::Platform interface file.
2. Simply put forwards to a concrete implementation (located anywhere, e.g. in Courier/Platform/Impl) of the platform interface in a header file, named like the corresponding platform interface file. E.g.

In order to get the (concrete or forwarded) platform implementation linked to the according Courier::Platform interface a namespace forward has to be done at the end of the header file of the real implementation as listed here:

```
namespace Courier { namespace Platform { namespace Impl {
    typedef Courier::Platform::Os::Generic::GenericMessageQueue MessageQueue;
}}}
```

> he only important thing with this namespace forward is the target namespace, which has to be Courier::Platform::Impl::<PlatformInterface>!

External Communication

External Communication Component

The external communication component (Courier::ExtCommComponent) is served as interface to the "outside world". One of Courier's basic concept are the Components, which are active entities communicating via messages among each other. Besides the core components, which represent the model view controller pattern, the external communication component incorporates this concept.

External communication in other words means interfacing the Courier world with information sent from and to non Courier modules. An ExtCommComponent plays the role of a gateway in this matter.

Like all other components, the ExtCommComponent is identified by an ID, the Component ID, which itself has to be unique per instance. Other than the dedicated Courier components, like ModelComponent, ViewComponent, ControllerComponent, a.s.o. the ExtCommComponent can be instantiated multiple times by providing this unique ID in its constructor interface.

Additionally the ExtCommComponent has an interface to attach/detach multiple external input handler and an interface to set one external output handler. The reason to have only one output handler per ExtCommComponent is that Courier messaging internally has no additional routing information for routing outgoing messages than the message tag "external". So Courier cannot differentiate between different types of "external" messages and sends all "external" Courier messages to all external output handler in the system.

```
    public:
        ExtCommComponent(ComponentId cid);
```

```
        virtual ~ExtCommComponent();

        void Attach(ExtCommInputHandler * extCommInputHandler);

        void Detach(ExtCommInputHandler * extCommInputHandler);

        IExtCommOutputHandler * SetExtCommOutputHandler(IExtCommOutputHandler * extCommOutputHar
```

> An ExtCommComponent may also be dedicated to external output handling respectively input handling only (just not attach an external input handler respectively set an external output handler).

A characteristic of Courier components is that they run in a (application defined) thread context defined by the component message receiver (Courier::ComponentMessageReceiver) they are attached to. In case of the ExtCommComponent this means, that all attached external input handler are running in this very same thread context. The external output handler assigned to an ExtCommComponent also runs in the same thread context receiving Courier messages marked with the message tag "external".

> In case an external input/output handler is blocking on its execution, the application designer is well advised to create a separate thread context (for listening from respectively distributing data to external modules), in order not to interfere any other external input/output handler or even component attached to the same ComponentMessageReceiver. An example for this can be found in the implementation of the Courier::Platform::ExtComm::PosixTerminalInputHandler.

External Input Handler

An external input handler is an object, which listens on an external communication channel. If an event on this channel is received, which shall be transported to the Courier system, the external input handler has to transform this message into an appropriate Courier message and post it into the Courier system. Courier provides a base (interface) class, which depicts the interface of an external input handler Courier is able to attach to an ExtCommComponent, this class is called Courier::ExtCommInputHandler.

An external input handler has to fulfil one simple interface, called Listen().

```
        static bool Post(Message * msg);

        virtual void Listen() = 0;
```

The static method Post() shall be used in the derived class of ExtCommInputHandler for posting Courier messages to the Courier system. It is also possible to simply use the method Post(), which is provided by the dedicated message itself. But using the Courier::ExtCommOutputHandler::Post() method, provides the possibility to monitor the events coming from external sources in a well-defined manner. The method Listen(), which has to be implemented in the respective external input handler, has to provide all the logic about extracting an event from an external source and transform it into a Courier message.

> The method Listen() is, as already mentioned above, running in the thread context of the component message receiver, where its ExtCommComponent is attached to. This method is triggered every loop cycle of the underlying thread, so a "forever"-loop must not be used in the implementation of the method Listen(). Running in the context of the component message receiver is also the reason not to use any blocking calls in this method.

External Output Handler

External output handler are simply defined by an interface in Courier, called Courier::IExtCommOutputHandler.

```
        virtual bool OnMessage(const Message & msg) = 0;
```

These output handler are application specific because they are simply used to transform a Courier message, which has to be tagged with "external", into a whatever corresponding external event.

```
        <message distribution="sequential" name="SpeedLimitViolationMsg" tag="external"
uid="{8628147C-CC61-4366-BC86-D1DD2C550B78}">
            <subscribers>
                <subscriber id="External" />
            </subscribers>
        </message>
```

Once an external output handler is implemented, for whatever reasons, it can be attached to any available ExtCommComponent in the system. Simply be aware of the above mentioned restrictions on blocking external input handler attached to an ExtCommComponent.

> The above message example also defines a dedicated subscriber (component) for the external message, which makes definitely sense for using the optimized routing path. The subscriber named "External" reflects the pre-defined Courier::ComponentType::ExternalCommunicator. This pre-defined component ID is still available in Courier, though it is not anymore assigned to an ExtCommComponent by Courier automatically. Reason is that the ExtCommComponent may be instantiated multiple times having now a Courier::ComponentId in its constructor. But the application can still re-use this pre-defined component ID or simply create its own customer specific component ID. Using a custom component ID means also that this custom component ID has to be used in the subscriber list of the message definition, if needed.

```
#include <Courier/Foundation/Component.h>


namespace CustomComponentId {
    enum Enum {
        WindowEventExtComm = Courier::ComponentType::CustomerFirst,
        TerminalEventExtComm
    };
}
```

> The blocking call limitations for external output handler are the same as for the external input handlers. Means if an external output handler uses blocking calls the corresponding ExtCommComponent resp. the underlying ComponentMessageReceiver are blocked too, which would result in not processing the messages in an appropriate manner. In this case a separate thread context is needed to resolve this issue.

How to Implement External Communication

External communication is in most cases dependent on the application requirements, as the application defines which events shall be received from and sent to the external modules. Additionally the external communication may differ on which platform configuration it is applied. E.g. on Win32 the window event handling will make use of the Win32 Messaging System, while on a Linux X11 environment the window events are represented by X11 event handling. Additionally it may be required, that besides window event handling another input source provides events, which

shall be forwarded to the Courier application (e.g. CAN, SPI, TCP/IP, Serial, etc.). For this reason is shall be possible to setup different external input handler or even external communication components, running in separate message receiver (thread contexts). Courier itself doesn't has a hard link to these application specific needs, but provides a framework to easily adopt to these needs. The translation from external event to Courier message and vice versa will basically always be part of the application, though Courier provides building blocks, which at least can provide a default transformation.

These building blocks, which are currently available, focus on external input handling. Courier simply provides this implementation, which are not linked into the Courier library but can be added to any application, that wants to make use of them.

Available pre-implemented External Input Handler

Courier already provides a small set of external input handler, which can be used either as building blocks in an applications platform specific code or as templates to implemented custom input handlers.

> These building blocks are not linked to the Courier library but may be linked into any application library (or executable). This enables the possibility to completely exchange external input handler for any need.

Currently following external input handler building blocks/templates are available in Courier source folder (see cgi_studio_courier/src/Courier/Platform/Impl/ExtComm/):

- Courier::Platform::ExtComm:Generic::GenericConsoleInputHander
- Courier::Platform::ExtComm:Posix::PosixTerminalInputHander
- Courier::Platform::ExtComm:Win32::Win32WindowInputHandler
- Courier::Platform::ExtComm:Wayland::WaylandInputHandler

While external input handler can be implemented in many variations, fulfilling the base class interface from Courier::ExtCommInputHandler, the mentioned Courier external input handler building blocks follow basically the same pattern. Every external input handler has a method Init() and if needed a method Dispose(), called at startup respectively shutdown phase of the applications platform environment. For transformation of an incoming event, these external input handler use a hook interface providing the (platform) specific event object. This (platform) specific hook interface can be used to implement the application specific event to Courier message transformation and assigned to the very external input handler.

> Currently all external input handlers provided by Courier (as application building blocks) are "non-blocking"! This has the benefit, that they will work in a single threaded application setup (e.g. all components running in the main thread).

> Using a non blocking external input handler has unfortunately a big drawback. When attached to an ExtCommComponent, which itself is running in a ComponentMessageReceiver together with other components, this ExtCommComponent will signal to be permanently executed (no wait). This may cause a boost in CPU usage respectively performance issues (e.g. for rendering)! To solve this problem the external input handler execution has to be throttled by either the Component throttling mechanism (available since Courier V2.1) or by simple add a wait state within the execution loop where the ComponentMessageReceiver of the respective ExtCommComponent is running.

Another solution to solve performance problems of "non-blocking" external input handler is to make them "blocking". Having a blocking external input handler implicates changes in the application's thread layout, which at least leads into having the external input handler run in its very own thread context. In Courier terms this means, maximum one external input handler is attached to an ExtCommComponent, which itself must be the only Component running in a ComponentMessageReveiver instance, that in turn is processed in a separate thread context. Though this seems a little restrictive it is the normal way how (external) input events are queried - by listen (and wait) on their respective input event queue in a separate thread. As this is the better approach for external input handling this topic is under review and most probably external input handler building blocks in blocking manner (wait and listen) will be supported.

An example for a Wayland input handler:

```
    class WaylandInputHandler : public ExtCommInputHandler {
        typedef ExtCommInputHandler Base;
    public:
        WaylandInputHandler();

        virtual ~WaylandInputHandler() {}

        bool Init(void * display, void * data, WaylandEventHook * eventHook = 0);

        WaylandEventHook * SetEventHook(WaylandEventHook * eventHook);

        virtual void Listen();

        const WaylandContext & GetContext() const { return mContext; }
```

```
    private:
        WaylandContext mContext;
    };
```

or an example of a POSIX terminal input handler,

```
    class PosixTerminalInputHandler : public ExtCommInputHandler {
        typedef ExtCommInputHandler Base;
    public:
        class Hook {
        public:
            virtual Message * OnEvent(Int event) = 0;
        };

        PosixTerminalInputHandler();

        bool Init
 (const Char * serialPortPath, bool mapStdIn, bool mapStdOut, bool mapStdErr, Hook * hook = 0);

        void Dispose();

        Hook * SetEventHook(Hook * hook);

        virtual void Listen();

        void OnEvent(Int event);

        void SetTerminalListenerName(const Char * name);

        const Char * GetTerminalListenerName() const;

    private:
        SerialPort mSerialPort;
        Hook * mHook;
        PosixTerminalListener mTerminalListener;
        CriticalSection mCs;
        bool mSerialPortOpened;
        bool mStdInMapped;
        bool mStdOutMapped;
        bool mStdErrMapped;
    };
```

which uses a separate thread for listening to terminal inputs:

```
    class PosixTerminalListener : public Thread {
    public:
        PosixTerminalListener();

        virtual ~PosixTerminalListener();

        bool Init(PosixTerminalInputHandler * inputHandler);
```

```
        void Terminate();

    protected:
        virtual Int ThreadFn();

    private:
        PosixTerminalInputHandler * mInputHandler;
        bool mTerminate;
    };
```

In order to get a building block linked to an application, simply add the sources to the application's build path. If Courier CMake system is used it looks like this:

```
CourierAddProjectFiles(
    ABSOLUTE_FILE_PATHS "."
    SOURCE_GROUP AppPlatform/${COURIER_PLATFORM_CONFIG}
    ${COURIER_SRC_DIR}/Courier/Platform/Impl/ExtComm/Win32/Win32WindowInputHandler.cpp
    ${COURIER_SRC_DIR}/Courier/Platform/Impl/ExtComm/Win32/Win32WindowInputHandler.h
)
```

Using Courier Provided External Input Handler

All external input handler building blocks rely basically on the same pattern: providing a hook interface which is used if available. If no hook object is applied to the respective external input handler, a default handling (if possible) may take place (e.g. sending key messages with keycode, when receiving a mappable key event). This means an application may re-use these building blocks without changing the code but simply applying a hook for events it wants to handle or overwrite (if the default handling is inappropriate).

See here an example of a hook implementation:

```
    class WindowInputEventHook : public
Courier::InputHandling::Win32::Win32WindowInputHandler::Hook
    {
    public:
        WindowInputEventHook() :
            mTouchEventPreprocessor(0),
            mRenderer(0) {}


        ~WindowInputEventHook()
        {
            mTouchEventPreprocessor = 0;
            mRenderer = 0;
        }
```

```
        void SetTouchEventPreprocessor(Courier::TouchHandling::TouchEventPreprocessor *
touchEventPreprocessor) {
            mTouchEventPreprocessor = touchEventPreprocessor;
        };

        virtual Courier::Message * OnEvent(const MSG & event);

        void SetRenderer(Courier::Renderer* renderer) { mRenderer = renderer; }

    private:
        Courier::TouchHandling::TouchEventPreprocessor * mTouchEventPreprocessor;
        Courier::Renderer* mRenderer;

    };
```

```
    Courier::Message * WindowInputEventHook::OnEvent(const MSG & event)
    {

        if (0 != mRenderer) {
            mRenderer->ForceKickDisplay();
        }

        Courier::Message * lMsg = Courier::InputHandling::InputHandler::cDoDefaultHandling;
        switch (event.message) {
        case WM_KEYDOWN:
            {
                COURIER_DEBUG_ASSERT(event.wParam <= 255);
                if (event.wParam == 'Q') {
                    lMsg = COURIER_MESSAGE_NEW(Courier::ShutdownMsg)();
                    // Post directly the message
                    Courier::InputHandling::InputHandler::Post(lMsg);
                }
                else if ((event.wParam >= '0') && (event.wParam <= '9')) {
                    const Courier::UInt32 lSpeed(static_cast<Courier::UInt32>((event.wParam -
'0') * 25));
                    COURIER_DEBUG_ASSERT(lSpeed <= 270);
                    lMsg = COURIER_MESSAGE_NEW(SpeedMsg)(lSpeed);
                }
                else if (event.wParam == 'A') {
                    lMsg = COURIER_MESSAGE_NEW(ToggleAutoPilotMsg)();
```

```
                }
                else if ((event.wParam == 'S') || (event.wParam == 'P') ||
                         (event.wParam == 'R') || (event.wParam == 'B') ||
                         (event.wParam == 'E') || (event.wParam == 'F')) {
                            const Courier::UInt8
lKeyCode(static_cast<Courier::UInt8>(event.wParam));
                            lMsg = COURIER_MESSAGE_NEW(TriggerAnimationActionMsg)(lKeyCode);
                }
                else if (event.wParam == 'O') {
                    lMsg = COURIER_MESSAGE_NEW(RenderStatisticsOverlayMsg)();
                }
                break;
            }
        case WM_MOUSEMOVE:
            // Don't handle moves without left button pressed
            if ((event.wParam & MK_LBUTTON) == 0) {
                break;
            }
            if(mTouchEventPreprocessor!=0) {
                if(mTouchEventPreprocessor-
>Receive(COURIER_MESSAGE_NEW(TouchMsg)(TouchMsgState::Move,
WINDOWS_GET_X_LPARAM(event.lParam), WINDOWS_GET_Y_LPARAM(event.lParam),
FeatStd::Internal::PerfCounter::Now(),static_cast<PointerId>(0),0))) {
                    lMsg = Courier::InputHandling::InputHandler::cIgnoreDefaultHandling;
                }
            }
            break;

        case WM_LBUTTONDOWN:
            COURIER_DEBUG_ASSERT((event.wParam & MK_LBUTTON) == MK_LBUTTON);
            if(mTouchEventPreprocessor!=0) {
                if(mTouchEventPreprocessor-
>Receive(COURIER_MESSAGE_NEW(TouchMsg)(TouchMsgState::Down,
WINDOWS_GET_X_LPARAM(event.lParam), WINDOWS_GET_Y_LPARAM(event.lParam),
FeatStd::Internal::PerfCounter::Now(),static_cast<PointerId>(0),0))) {
                    lMsg = Courier::InputHandling::InputHandler::cIgnoreDefaultHandling;
                }
            }
            break;
```

```
        case WM_LBUTTONUP:
            COURIER_DEBUG_ASSERT((event.wParam & MK_LBUTTON) == 0);
            if(mTouchEventPreprocessor!=0) {
                if(mTouchEventPreprocessor-
>Receive(COURIER_MESSAGE_NEW(TouchMsg)(TouchMsgState::Up, WINDOWS_GET_X_LPARAM(event.lParam),
WINDOWS_GET_Y_LPARAM(event.lParam),
FeatStd::Internal::PerfCounter::Now(),static_cast<PointerId>(0),0))) {
                    lMsg = Courier::InputHandling::InputHandler::cIgnoreDefaultHandling;
                }
            }
            break;

        default:
            break;
        }

        return lMsg;
    }
```

This window event hook then has to be linked to an external input handler

```
// Configure external communication input handling
lRc = lRc && mWin32WindowInputHandler.Init(handle, &mWindowInputEventHook);
COURIER_DEBUG_ASSERT(lRc);
```

and the external input handler has to be attached to an external communication component.

```
// Attach external input handler to external communication component
mExtCommComponent.Attach(&mWin32WindowInputHandler);
```

Using External Output Handler

As already mentioned more than one external input handler may be attached to one ExtCommComponent. Additionally to external input handlers an external output handler may be assigned to the ExtCommComponent. See here a sample of a very simple external output handler:

```
#include <Courier/Foundation/IExtCommOutputHandler.h>

class ExtCommOutputHandler : public Courier::IExtCommOutputHandler
{
public:
    ExtCommOutputHandler() {}
```

```
    virtual bool OnMessage(const Courier::Message & message);
};
```

```
bool ExtCommOutputHandler::OnMessage(const Courier::Message & message)
{
    printf("***** %s processed by external entity *****\n", message.GetName());
    return true;
}
```

This external output handler may also be assigned to the same ExtCommComponent (but can also be assigned to a different one).

```
    // Configure platform independent external communication component for output handling
    (void) mExtCommComponent.SetExtCommOutputHandler(&mExtCommOutputHandler);
```

> The external communication component which an external output handler is assigned to is the one which shall be assigned in an Courier "external" tagged message's subscriber list. If an "external" tagged message has no subscriber list it is assumed to be "broadcasted", that is this message is sent to all ExtCommComponents having an external output handler attached.

Starting an External Communication Component

We already learned that the ExtCommComponent has to be instantiated with an unique component ID, which can either be the one available from Courier::ComponentType, called ExternalCommunicator or any custom defined component ID.

```
    mExtCommComponent(Courier::ComponentId(Courier::ComponentType::ExternalCommunicator))
```

Once an external communication component has its external input and/or output handler attached (Using Courier Provided External Input Handler resp. Using External Output Handler), it is ready to be linked to the Courier application.

Next step is to attach the ExtCommComponent to a ComponentMessageReceiver, which provides (beside other things like message queue, etc.) the thread context in which the ExtCommComponent is supposed to run. A few ExtCommComponents may have special requirements, like an ExtCommComponent for window event handling,

```
    // Attach platform dependent window event handler
    mMsgReceiver.Attach(mAppEnvironment->GetWindowEventExtCommComponent());
```

others may run in a separate thread context.

```
    // Start external communicator in separate thread
    ComponentMessageReceiverThread lExternalCommunicatorThread;
    lExternalCommunicatorThread.SetName("ExtComm");
    lExternalCommunicatorThread.Attach(mExtCommComponent);
    lExternalCommunicatorThread.Activate();
    rc = rc && lExternalCommunicatorThread.Run();
```

The above piece of code uses the class ComponentMessageReceiverThread. This class is not part of Courier but simply a helper class to let a ComponentMessageReceiver run in a separate thread. Here is the complete implementation of this basic helper:

```
class ComponentMessageReceiverThread : public Courier::Platform::Thread
{
    COURIER_LOG_SET_REALM(LogRealm::SampleApp);
public:
    class Hook {
        public:
            virtual ~Hook() {}
            virtual bool OnStartup(void * data) { FEATSTD_UNUSED(data); return true; }
            virtual bool OnExecute(void * data) { FEATSTD_UNUSED(data); return true; }
            virtual void OnShutdown(void * data, bool normalShutdown) { FEATSTD_UNUSED2(data,
normalShutdown); }
    };

    ComponentMessageReceiverThread(Hook * hook = 0, void * data = 0) : mHook(hook),
mData(data) {}

    Hook * SetHook(Hook * hook) {
        Hook * lPrevHook = mHook;
        mHook = hook;
        return lPrevHook;
    }

    void * SetData(void * data) {
        void * lPrevData = data;
        mData = data;
        return lPrevData;
```

```
    }

    // Make receiver active to start receiving messages
    void Activate() {
        mMsgReceiver.Activate();
    }

    void Attach(Courier::Component & component) {
        mMsgReceiver.Attach(&component);
    }

    void Detach(Courier::Component & component) {
        mMsgReceiver.Detach(&component);
    }

    void SetCycleTime(Courier::ComponentMessageReceiverTiming::Enum type, FeatStd::UInt32
timeMs) {
        mMsgReceiver.SetCycleTime(type, timeMs);
    }

    FeatStd::UInt32 GetCycleTime() const { return mMsgReceiver.GetCycleTime(); }


protected:
    virtual Courier::Int ThreadFn() {
        mMsgReceiver.SetName(this->GetName());

        bool lSuccess = true;
        if (0 != mHook) {
            lSuccess = lSuccess && mHook->OnStartup(mData);
        }

        // The message processing loop
        while (lSuccess && !mMsgReceiver.IsShutdownTriggered()) {
            if (0 != mHook) {
                lSuccess =  mHook->OnExecute(mData);
            }
            lSuccess = lSuccess && mMsgReceiver.Process();
        }
```

```
        if (0 != mHook) {
            mHook->OnShutdown(mData, lSuccess);
        }

        Courier::Int lRc;
        if (!lSuccess) {
            COURIER_LOG_FATAL("Thread (%s) processing failed!", GetName());
            lRc = -1;
        } else {
            COURIER_LOG_INFO("Thread (%s) processing ended normal.", GetName());
            lRc = 0;
        }

        return lRc;
    }

private:
    Courier::ComponentMessageReceiver mMsgReceiver;
    Hook * mHook;
    void * mData;
};
```

After attaching the components to its appropriate message receiver, all message receiver have to be activated, which basically means, that they are registered to the Courier internal message router. The message router now know all the message receivers (respectively their message queues) in the system and is able to forward the messages, based on their characteristics (distribution strategy, tags, etc.). Sending the Courier::StartupMsg, which shall always be the very first message to be sent, informs all components about the system startup.

```
    // Before sending the start up (resp. the first) message be sure to activate all message
receivers
    mMsgReceiver.Activate();

    // Startup the system by posting the startup message after all components are in place
    Courier::Message * msg = COURIER_MESSAGE_NEW(Courier::StartupMsg)();
    lRc = lRc && (msg!=0 && msg->Post());
    COURIER_DEBUG_ASSERT(lRc);
```

Afterwards let the message receiver being processed in the "main" thread context, if not already done via a ComponentMessageReceiverThread (see code sample above).

```
        bool lSuccess;
        do {
            // this is the main message processing loop.
            lSuccess = mMsgReceiver.Process();
/*
            FeatStd::UInt32 fps2D = mViewHandler.GetFramesPerSecond2D();
            FeatStd::UInt32 fps3D = mViewHandler.GetFramesPerSecond();
            printf("FPS 2D(%u) 3D(%u) \n",fps2D,fps3D);
            COURIER_UNUSED2(fps2D,fps3D);
*/
        } while (lSuccess && !mMsgReceiver.IsShutdownTriggered());

        if (!lSuccess) {
            COURIER_LOG_FATAL("Main loop processing failed!");
        } else {
            COURIER_LOG_INFO("Main loop processing ended normal.");
        }
        lRc = lSuccess;
```

Graceful Termination of External Communication Component

Cleaning up has to be done to the external communication component to terminate a program gracefully. Maybe the most important is to call the external input handlers Dispose() method, if one exists, which may restore respectively free external resources.

For example in the implementation of the PosixTerminalInputHandler, the Dispose() method restores the terminal settings, which were changed in initialization phase.

```
        // Cleanup external communication input handling
        mExtCommComponent.Detach(&mTerminalInputHandler);
        mTerminalInputHandler.Dispose();
```

```
    // Cleanup external communication output handling
    (void) mExtCommComponent.SetExtCommOutputHandler(0);
```

The above code fragments perform detach of the external input handler applied on system shutdown and reset the assigned external output handler.

>

> The application shall always care to cleanup and restore all the resources (like threads, external resources) it has used during its execution to avoid a mess up of the platform.

E.g. gracefully shutdown of component message receiver threads after Courier::ShutdownMsg was issued:

```
        toggleThread.Stop();
        do {
            lTerminate = true;
#if (0 != ENABLE_VIEW_THREAD)
            lTerminate = lTerminate && (Courier::Platform::ThreadStatus::Terminated ==
lViewThread.GetStatus());
#endif
#if (0 != ENABLE_RENDER_THREAD)
            lTerminate = lTerminate && (Courier::Platform::ThreadStatus::Terminated ==
lRenderThread.GetStatus());
#endif
#if (0 != ENABLE_CONTROLLER_THREAD)
            lTerminate = lTerminate && (Courier::Platform::ThreadStatus::Terminated ==
lControllerThread.GetStatus());
#endif
#if (0 != ENABLE_MODEL_THREAD)
            lTerminate = lTerminate && (Courier::Platform::ThreadStatus::Terminated ==
lModelThread.GetStatus());
#endif
#if (0 != ENABLE_EXTCOMM_THREAD)
            lTerminate = lTerminate && (Courier::Platform::ThreadStatus::Terminated ==
lExtCommThread.GetStatus());
#endif
#if defined(COURIER_IPC_ENABLED)
            lTerminate = lTerminate && (Courier::Platform::ThreadStatus::Terminated ==
lIPCThread.GetStatus());
#endif
            lTerminate = lTerminate && (Courier::Platform::ThreadStatus::Terminated ==
toggleThread.GetStatus());

        } while (!lTerminate);
```

Message Factories

Usually Courier objects are created/deleted using the macros COURIER_NEW() and COURIER_DELETE().

For customization reasons Courier::Message instances must be handled in a different way.

Therefore a platform specific interface (Courier::Platform::MessageFactory) was introduced:

```
    Message* lpMessage;
    lpMessage = COURIER_MESSAGE_NEW(Helper::BroadcastMsg)();
    EXPECT_NE((void*)0, lpMessage);

    MessageFactory::Destroy(lpMessage);
```

For convenience reasons the macro COURIER_MESSAGE_NEW() was defined.

There are two different ways of Message allocation: **static** and **dynamic**. This is reflected by either using DynamicMessageFactory or StaticMessageFactory.

DynamicMessageFactory

Courier::Platform::Messaging::DynamicMessageFactory allows the creation of an unlimited number of Message instances using the Courier memory management.
See Candera::MemoryManagement::PlatformHeap for details.

StaticMessageFactory

Courier::Platform::Messaging::StaticMessageFactory preallocates memory for a certain number of Message instances using Courier::StaticObjectBuffer. The **number of instances** must be requested by using Courier::StaticObjectBuffer_Private::BufferData specialization. Please note that the specialization has to be done either in global namespace or in namespace Courier (prefered).

```
class TestClass_A { };
static FeatStd::SizeType sBuffer_A[((sizeof(TestClass_A) + SIZE_OF_VOID_PTR + SIZE_OF_VOID_PTR
- 1) / SIZE_OF_VOID_PTR) * 8];
    static const Courier::Internal::StaticObjectBuffer_Private::BufferDataTyped<TestClass_A>
sBuffer_AT(8, sBuffer_A);

namespace Courier {
    class TestClass_A { };
    static FeatStd::SizeType sBuffer_A[((sizeof(TestClass_A) + SIZE_OF_VOID_PTR +
```

```
SIZE_OF_VOID_PTR - 1) / SIZE_OF_VOID_PTR) * 3];
    static const Courier::Internal::StaticObjectBuffer_Private::BufferDataTyped<TestClass_A>
sBuffer_AT(3, sBuffer_A);
```

For convenience reasons the macro COURIER_MESSAGE_CONFIGURATION() was defined:

```
#include "MessageFactoryConfiguration.h"

namespace AppPlatform {
    void LoadMessageFactoryConfiguration()
    {
    }

} // namespace AppPlatform

namespace Courier {
    COURIER_MESSAGE_CONFIGURATION( ::Courier::Internal::ListEventMsg, 10);

// ------------------------------------------------------------------------
    COURIER_MESSAGE_CONFIGURATION( ::Courier::UpdateModelMsg, 10);

// ------------------------------------------------------------------------
    COURIER_MESSAGE_CONFIGURATION( ::Courier::StartupMsg, 1);
```

See CourierSampleApp/AppPlatform/MessageFactoryConfiguration.cpp for details.

For configuring the usage of StaticMessageFactory please see chapter Platform Implementation.

> If CourierGenerator.exe is used to generate MessageFactoryConfiguration.cpp the configuration has to be done in XML (MessageFactoryConfiguration.xcmdl).

> Workaround: Due to the fact that some linkers throw away unreferenced static declarations, MessageFactoryConfiguration.[h|cpp] has to implement a method which has to be called before the first Message creation. In SampleApp this is done in AppEnvironment::Setup().

OS Abstraction

FeatStd provides ready-to-use implementations based on different OS, environments, etc., which can be used as a construction kit for any platform setup. These sources are located in the folder

```
./featstd/src/FeatStd/Platform/
```

Various operating systems implementations are available, like Win32, Posix, Integrity, etc. and Generic, which basically implement the interface based on other interfaces (e.g. Mutex, CriticalSection, AtomicOp, etc.) or have stub implementations for atomic platform interfaces (means they **must** be implemented for the target OS), like Semaphore, Thread, Ticks, etc.

Revision #13
Created 2023-02-20 05:30:17 UTC by Tsuyoshi.Kato
Updated 2026-08-18 04:47:25 UTC by Tsuyoshi.Kato