Serialization and Corruption Detection of Messages

Description

Courier provides a mechanism which allows serialization of messages. The serialization mechanism is only available when using the feature FEATSTD_IPC_ENABLED. Serialized messages can then be sent from IPC component of application A to the IPC component of application B.

Additionally the serialization of messages can be used for computing a checksum and using this checksum for checking if the content of the messages got corrupted. This is especially necessary when using interprocess communication but also make sense for message distribution inside one address space.

Serialization of Messages

Customer has just to define a message with the attribute **serializable="true"** . The message generator then generates necessary meta info into the message source code which is used by the serialization mechanism.

For example for the message ToMaudeControllerPayloadIpcMsg:

```
    <message name="ToMaudeControllerPayloadIpcMsg" serializable="true" distribution="sequential"
      <subscribers>
        <subscriber id="Controller" process="Maude" />
      </subscribers>
      <member name="FixedSizeStringPayload" type="FixedSizeStringPayload" uid="{073e7195-a06e-47
      <member name="Int32Payload" type="Int32Payload" uid="{a09b107f-a4a9-4eaa-a9af-fb22ef93414b
    </message>
```

the following serialization meta data will be generated:

```
#if defined COURIER_IPC_ENABLED
const ::Courier::SerializationInfo * ToMaudeControllerPayloadIpcMsg::GetSerializationInfo() cons
{
    static const ::Courier::SerializationMemberInfo gSerializationInfo[] = {
        { &::Courier::Serialization< ::Courier::SerializationTypeTrait< FixedSizeStringPayload >
SERIALIZATION_OFFSET(ToMaudeControllerPayloadIpcMsg, mFixedSizeStringPayload) },
        { &::Courier::Serialization< ::Courier::SerializationTypeTrait< Int32Payload >::Type >,
SERIALIZATION_OFFSET(ToMaudeControllerPayloadIpcMsg, mInt32Payload) },
    };
```

```
    static const ::Courier::DeserializationMemberInfo gDeserializationInfo[] = {
        { &::Courier::Deserialization< ::Courier::SerializationTypeTrait< FixedSizeStringPayloac
SERIALIZATION_OFFSET(ToMaudeControllerPayloadIpcMsg, mFixedSizeStringPayload) },
        { &::Courier::Deserialization< ::Courier::SerializationTypeTrait< Int32Payload >::Type >
SERIALIZATION_OFFSET(ToMaudeControllerPayloadIpcMsg, mInt32Payload) },
    };
    static const ::Courier::SerializationInfo gInfo = {
        Base::GetSerializationInfo(),
        gSerializationInfo,
        gDeserializationInfo,
        2
    };
    return &gInfo;
}
#endif
```

As one can see the message generator generates an entry for each member of the message and uses a template for the serialization of each type. If a specialized template for a certain type exists the specialized template will be used for serialization, otherwise the Courier default implementation.

Customer Type Serialization

Serialization of Messages

Imagine you have a class or a struct which have the following members:

```
class CustomerClass
{
    public:
        friend FeatStd::UInt32 Courier::Serialization< CustomerClass >(const void * elemPtr,
void * byteDest);
        friend FeatStd::UInt32 Courier::Deserialization< CustomerClass >(const void * elemPtr,
void * byteDest);

        enum { StringLength = 128 };

        CustomerClass() : mVal(0), mVal2(0), mCustomerEnum(EnumerationValue1) { mCharArray[0]
= 0; }
        CustomerClass(const CustomerClass & c) : mVal(c.mVal),mVal2(c.mVal2),
mCustomerEnum(c.mCustomerEnum) {
            FeatStd::Internal::String::CopyPartial(mCharArray,c.mCharArray,StringLength-1);
            mCharArray[StringLength-1] = 0;
```

```
        }

        CustomerClass(FeatStd::Int val, CustomerEnum enumVal, const FeatStd::Char * str) :
mVal(val),mVal2(val),mCustomerEnum(enumVal) {
             FeatStd::Internal::String::CopyPartial(mCharArray,str,StringLength-1);
             mCharArray[StringLength-1] = 0;
        }

        bool operator==(const CustomerClass & s) const {
            return s.mVal==mVal && s.mVal2==mVal2 &&
0==FeatStd::Internal::String::CompareStrings(mCharArray,s.mCharArray) && s.mCustomerEnum==
mCustomerEnum;
        }

    private:
        FeatStd::Int mVal;
        FeatStd::Char mCharArray[StringLength];
        FeatStd::Int mVal2;
        CustomerEnum mCustomerEnum;
};

namespace Courier {
    template <> FeatStd::UInt32 Serialization< CustomerClass >(const void * elemPtr, void *
byteDest);
    template <> FeatStd::UInt32 Deserialization< CustomerClass >(const void * byteSource, void
* elemPtr);
}
```

You only have to add the Serialization and Deserialization template functions for this CustomClass type.

The implementation of those functions (this is just an example and shows that you also can reuse the already existing basic data type serialization functions) may look like this:

```
namespace Courier {

    template <> FeatStd::UInt32 Serialization< CustomerClass >(const void * elemPtr, void *
byteDest)
    {
        const CustomerClass * c = reinterpret_cast<const CustomerClass*>(elemPtr);
```

```
        UInt32 written = Serialization<Int>(&(c->mVal),(byteDest==0)?0:Courier::ToUInt8Ptr(byteDest));
        written += Serialization<TChar*>(&(c->mCharArray),(byteDest==0)?0:Courier::ToUInt8Ptr(byteDest,written));
        written += Serialization<Int>(&(c->mVal2),(byteDest==0)?0:Courier::ToUInt8Ptr(byteDest,written));
        written += Serialization<CustomerEnum>(&(c->mCustomerEnum),(byteDest==0)?0:Courier::ToUInt8Ptr(byteDest,written));
        return written;
    }

    template <> FeatStd::UInt32 Deserialization< CustomerClass >(const void * byteSource, void * elemPtr)
    {
        CustomerClass * c = reinterpret_cast<CustomerClass*>(elemPtr);

        UInt32 read = Deserialization<Int>(Courier::ToUInt8Ptr(byteSource),&(c->mVal));
        read += Deserialization<TChar*>(Courier::ToUInt8Ptr(byteSource,read),&(c->mCharArray));
        read += Deserialization<Int>(Courier::ToUInt8Ptr(byteSource,read),&(c->mVal2));
        read += Deserialization<CustomerEnum>(Courier::ToUInt8Ptr(byteSource,read),&(c->mCustomerEnum));
        return read;
    }
}
```

If you want to use that CustomerClass inside a message just define the message the usual way:

```
   <message uid="{fae2602f-d04a-4fd2-b98d-3b3569e8f1bf}" name="CustomClassMsg" serializable="tru
      <subscribers>
        <subscriber id="Controller" />
      </subscribers>
      <member uid="{909323b4-029d-42ba-aed2-fc39619150b3}" name="P1" type="CustomerClass" />
      <member uid="{372324d9-a396-4bc3-ac9d-7488d343e966}" name="P2" type="CustomerClass" />
    </message>
```

The code generated will be as following and will be used whenever the CustomClassMsg gets serialized:

```
#if defined COURIER_IPC_ENABLED
const ::Courier::SerializationInfo * CustomClassMsg::GetSerializationInfo() const
{
```

```
    static const ::Courier::SerializationMemberInfo gSerializationInfo[] = {
        { &::Courier::Serialization< ::Courier::SerializationTypeTrait< CustomerClass >::Type >,
SERIALIZATION_OFFSET(CustomClassMsg, mP1) },
        { &::Courier::Serialization< ::Courier::SerializationTypeTrait< CustomerClass >::Type >,
SERIALIZATION_OFFSET(CustomClassMsg, mP2) },
    };
    static const ::Courier::DeserializationMemberInfo gDeserializationInfo[] = {
        { &::Courier::Deserialization< ::Courier::SerializationTypeTrait< CustomerClass >::Type
SERIALIZATION_OFFSET(CustomClassMsg, mP1) },
        { &::Courier::Deserialization< ::Courier::SerializationTypeTrait< CustomerClass >::Type
SERIALIZATION_OFFSET(CustomClassMsg, mP2) },
    };
    static const ::Courier::SerializationInfo gInfo = {
        Base::GetSerializationInfo(),
        gSerializationInfo,
        gDeserializationInfo,
        2
    };
    return &gInfo;
}
#endif
```

Multidimensional Arrays

There is currently no "automated" way to serialize/deserialize multidimensional arrays. We recommend implementing your own struct or class which wraps (implements) the multidimensional array.

Additionally you have to implement the Serialization/Deserialization template function for this new type like described in Serialization of Messages.

Message Corruption Handling

The serialization mechanism is also used for detecting corrupted messages.

This means that when a message is posted checksums are computed and this checksums are then part of the message. Whenever the OnMessage methods of components are called the checksums are computed again and compared with the checksums of the message. If a corruption is detected the optional callback Courier::Component::MessageCorruptionFct might be called.

The implementation of this callback might look like this:

```
static void HaroldAppMessageCorruptionHandler(Component * component, const Message & message)
{
```

```
    if(component!=0) {
        FEATSTD_LOG_ERROR("Message corruption occured when sending a message to Component %u
Message 0x%08x",component->GetId(),message.GetId());
    }
}
```

Setting of the callback is done during application initialization by calling the method Courier::Component::SetMessageCorruptionCallback:

```
    Component::SetMessageCorruptionCallback(&HaroldAppMessageCorruptionHandler);
```

In the HaroldApp test application the corruption is simulated by posting a message and afterwards changing a byte.

```
#ifdef PERFORM_MESSAGE_CORRUPTION_TEST
                msg = COURIER_MESSAGE_NEW(CorruptionTestMsg)(0x11111111);
                FEATSTD_DEBUG_ASSERT(msg!=0);
                rc = msg->Post();
                // corrupt it
                UInt8 * buffer = ToUInt8Ptr(msg);
                buffer[18] = 0x21;
                if(rc) {
                    FEATSTD_LOG_INFO("Sent CorruptionTestMsg");
                }
#endif
```

Revision #6
Created 2023-02-27 05:20:52 UTC by Tsuyoshi.Kato
Updated 2025-06-01 22:19:56 UTC by Elisabeth Zauner