Satellite Control Software (SCS)

Mission Control System Extensibility

User Guide

Prepared by:

Florian George

Stéphane Billeter

•

Space Center EPFL

Lausanne

Switzerland

•

06 November 2013

•

RECORD OF REVISIONS

ISS/REV	Date	Modifications	Created/modified by
1/0	04/08/2011	Initial release	Florian George
2/0	06/11/2013	Updated for QB50. Release ICD for CDR.	Stéphane Billeter / Muriel Richard

ABBREVIATED TERMS

Ack	Acknowledgement
CCSDS	Consultative Committee for Space Data Systems
CRD	Custom Reporting Data
COM	Component Object Model
ECSS	European Cooperation for Space Standardization
EGSE	Electrical Ground-Support Equipment
ESA	European Space Agency
MCS	Mission Control System
MDR	Mission Data Repository
Nack	Negative Acknowledgment
OSI	Open Systems Interconnection
PUS	Packet Utilisation Standard (see [N1])
SCOE	Special Checkout Equipment
TC	Telecommand
TM	Telemetry
VC	Virtual Channel

Introduction

This document describes the extensibility infrastructure of the Mission Control System (MCS). The MCS has two extensibility points: at processing and at distribution stages. Figure 1 shows the simplified data flow of the built-in and custom reporting data in the MCS during processing and distribution stages. As shown, custom modules can be inserted at both the processing and distribution stages. They allow the MCS to process packets and distribute data for mission-specific or unsupported ECSS-E-70-41A services. The Mission Data Client offers a similar extensibility that allows client modules to retrieve and display the custom reporting data; this is described in [N2].

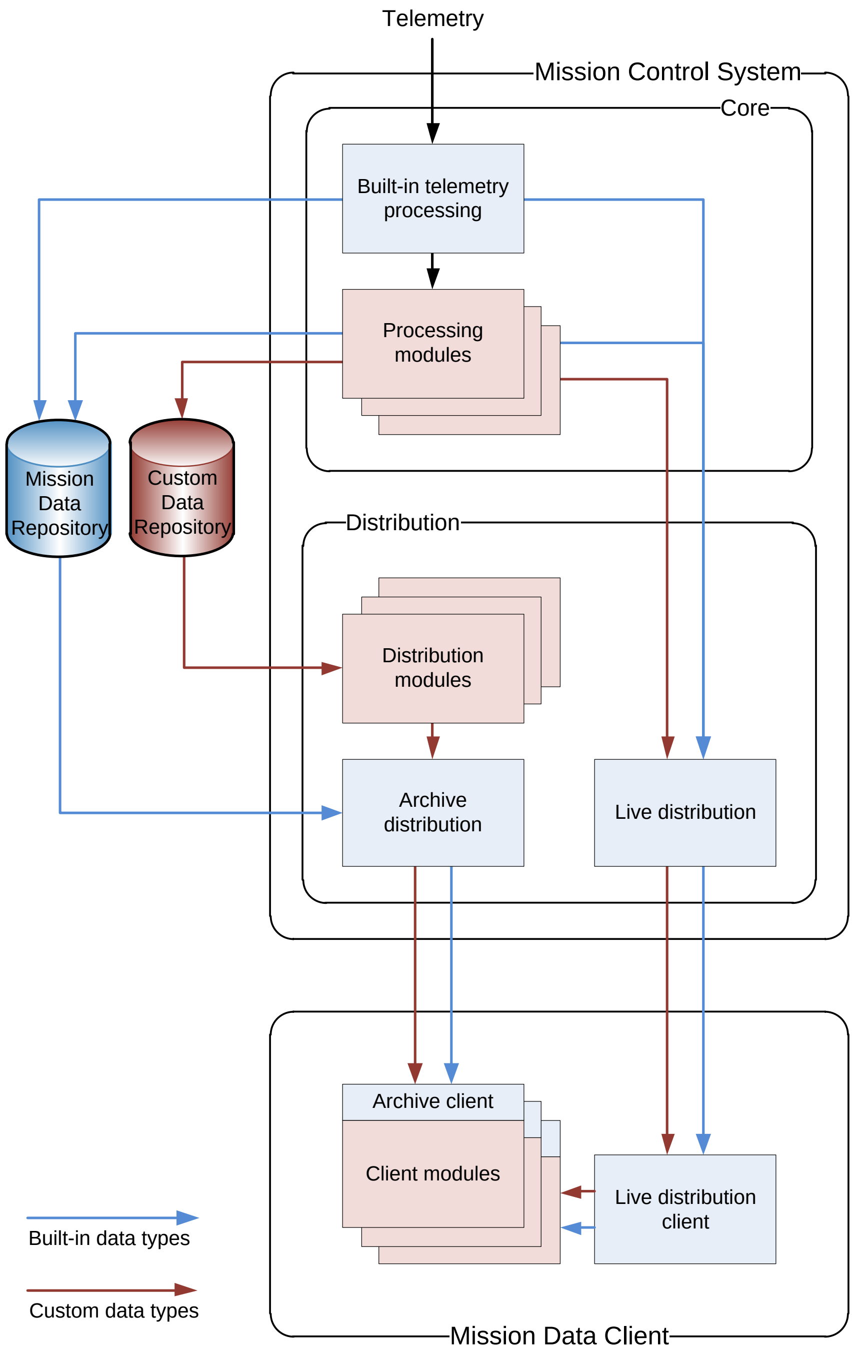


Figure 1 – Simplified data flow of the built-in and custom reporting data in the MCS.

As the main role of a distribution module is to retrieve the data archived by the corresponding processing module, they shall be developed in pairs. Also note that while the built-in Mission Data Repository and the Custom Data Repository are logically distinct, they can be part of the same physical storage; e.g. additional tables in the same database.

The MCS is entirely built using the .NET Framework[1], modules are no exception. A module is a class implementing a specific interface and configured to be loaded by the MCS. Multiple modules can be present in a single assembly. Chapter 2 describes the processing modules in details; Chapter 3 the distribution modules.

Modules are instantiated at MCS's start-up and reused for each telemetry packet (in case of processing modules) or custom reporting data archive request (for distribution modules).

While a module can theoretically contain or call native code, it is not recommended. In such case, the stability and integrity of the MCS could not be ensured anymore (modules are loaded in-process in the current version of the MCS).

[1] .NET Framework: http://www.microsoft.com/net/

1 CUSTOM REPORTING DATA

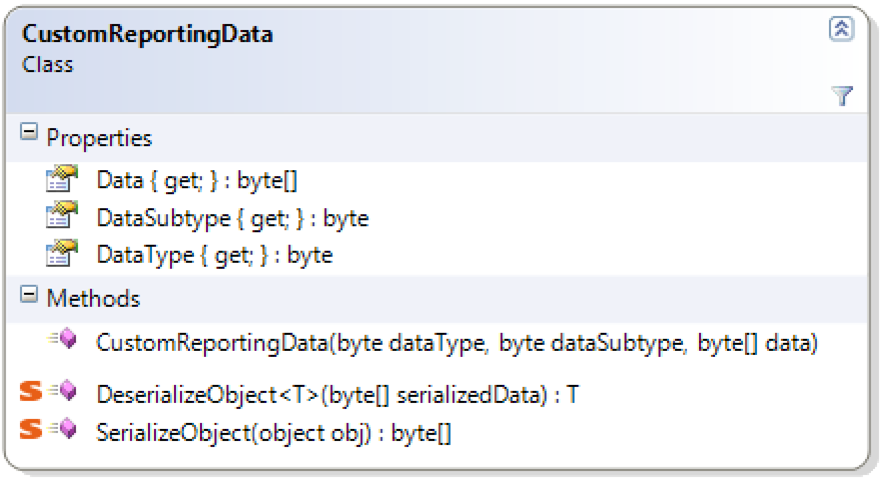
Processing modules can generate data for a non-built onboard service, for mission-specific onboard services or even data specific to the module itself. This data can be stored for archiving and/or sent to clients through the Live Distribution. In complement, distribution modules retrieve this archived data on demand to make it available to clients. This data is called Custom Reporting Data (CRD) in the MCS.

CRD is always exposed and transferred in the form of instances of the *CustomReportingData* class. This class, as shown in Figure 2, is a simple container pairing the data and the data type and subtype.

The data must be binary-serializeable (convertible to a byte array). Two static methods to serialize and deserialize data are present to provide a default binary serialization mechanism based on the .NET Framework's *DataContractSerializer*[2]. Any other reversible [on the client-side] mechanism resulting in a byte array can be used instead.

A data type is similar to a service type of a PUS services in [N1]. For consistency, it is recommended that the data type used by the processing and distribution modules is the same as the PUS service generating the data. For example, if the spacecraft implement a mission-specific service type 128 that generate images, the modules would expose the images and associated data as data type 128.
Subtypes can be freely used for the different subtypes of data generated/stored. For example, in the case the service generating images, there could a subtype for the images themselves, for the metadata, for a summary of the transmitted images, etc.

.


CustomReportingData
Class

Properties
Data { get; } : byte[]
DataSubtype { get; } : byte
DataType { get; } : byte

Methods
CustomReportingData(byte dataType, byte dataSubtype, byte[] data)
DeserializeObject<T>(byte[] serializedData) : T
SerializeObject(object obj) : byte[]

Figure 2 - *CustomReportingData* class

[2] DataContractSerializer: http://msdn.microsoft.com/en-us/library/ms405768.aspx

2 PROCESSING MODULES

Processing modules are loaded into the Core process of the MCS. Every valid telemetry packets received by the MCS is handed to each loaded processing module. They can store built-in data in the Mission Data Repository (see chapter 2.3 *Store built-in data in Mission Data Repository*) and can generate live custom reporting data (see chapter 2.4 *Generating Custom Reporting Data*).

The role of processing modules is to:

- Process *source data* contained in telemetry packets
- Generate built-in data (housekeeping values and acknowledgments) if applicable
- Generate CRD and store in a custom data repository of the module's choice

To be loaded into the MCS's Core, the subsequent steps must be followed:
- The module's assembly must be placed in the *Modules* subdirectory of the MCS installation directory.
- The module's type reference must be present in the configuration file of the MCS's Core executable. See chapter 3.1 *Configuring MCS to load distribution modules.*
- The type specified at the previous step must be a class implementing the *IProcessingModule* interface. See chapter 2.2 *IProcessingModule interface.*

Important note: do not place *GroundSegment.*.dll* or *Mcs.*.dll* assemblies in the *Modules* subdirectory. Otherwise they will exist in both the *MCS* directory and the *Modules* subdirectory and modules may fail to load. Only place in the *Modules* subdirectory the assemblies of the modules and any other assemblies specific to your modules.

2.1 Configuring MCS to load processing modules

To instruct the MCS's Core to load a processing module, its configuration shall be updated. The configuration file is the XML file named *Core.exe.config* and located in the installation directory of the MCS. An XML element matching the following format must be added to the *processingModules* section:

```
<add type="assembly-qualified-type-name" />
```

The *type* attribute is the assembly-qualified name[3] of the type implementing the *IProcessingModule* interface. Here is an example of the *processingModules* section configured to load two processing modules:

```
<processingModules>
  <add type="SwissCube.Housekeeping.ProcessingModule, HousekeepingProcessingModule" />
  <add type="SwissCube.ImageService.ProcessingModule, ImageServiceProcessingModule" />
</processingModules>
```

Figure 3 - Example of *processingModules* configuration section with two processing modules

[3] .NET Framework assembly-qualified type names:
http://msdn.microsoft.com/en-us/library/system.type.assemblyqualifiedname.aspx

2.2 IProcessingModule interface

The *IProcessingModule* interface is defined in the *Mcs.Core.Contracts* assembly and resides in the *Epfl.SwissCube.GroundSegment.Mcs.Core* namespace.

```
using System;
using Epfl.SwissCube.GroundSegment.Data.Ccsds;

namespace Epfl.SwissCube.GroundSegment.Mcs.Core
{
   // Defines the interface implemented by the processing modules.
   public interface IProcessingModule
   {
      // Must return the name of the processing module.
      string Name { get; }

      // Called by the MCS when initializing the module.
      //  missionDataAccess: Reference to a Mission Data Access service.
      void Initialize(IMissionDataAccess missionDataAccess);

      // Called by the MCS when unloading the module.
      void Unload();

      // Process the telemetry packet.
      //  telemetry:      The telemetry packet to be processed by the module.
      //  spacecraftTime: Correlated on-board time.
      //  spacecraftInfo: The spacecraft info associated with the telemetry.
      void ProcessTelemetry( Telemetry telemetry,
                             DateTime spacecraftTime,
                             ISpacecraftInfo spacecraftInfo);
   }
}
```

Figure 4 - *IProcessingModule* interface to be implemented by the processing modules.

2.3 Store built-in data in Mission Data Repository

During initialization, the module has its *Initialize* method called with a reference to an object implementing the *IMissionDataAccess* interface. Through the interface, the module can store data such as housekeeping values and acknowledgements in the Mission Data Repository. This data will be automatically distributed through the Live Distribution as well.

2.4 Generating Custom Reporting Data

The persistent storage, if applicable, of the Custom Reporting Data generated by a processing module is up to the module. However, the stored data must also be accessible by the corresponding distribution module for Archive Distribution. See chapter 1 *Custom Reporting Data* for details about the CRD itself.

The distribution of the CRD through the Live Distribution is done using the *SendCustomReportingDataToDistribution* method of the *IMissionDataAccess* interface passed at initialization of the module.

3 DISTRIBUTION MODULES

Processing modules are loaded into the Distribution process of the MCS. They allow on-demand (archive distribution) access from clients to the custom reporting data processed and stored by the processing module.

The role of the distribution modules is to:

- Enable clients to access the archived custom reporting data stored by a processing module

Each distribution module is associated with a single data type (see chapter 1 *Custom Reporting Data* for details about the CRD data types).

To be loaded into the MCS's Distribution, the subsequent steps must be followed:

- The module's assembly must be placed in the *Modules* subdirectory of the MCS installation directory.
- The module's type reference must be present in the configuration file of the MCS's Distribution executable. See chapter **Error! Reference source not found.** ***Error! Reference source not und.***.
- The type specified at the previous step must be a class implementing the *IDistributionModule* interface. See chapter 3.2 *IDistributionModule interface*.

Important note: do not place *GroundSegment.*.dll* or *Mcs.*.dll* assemblies in the *Modules* subdirectory. Otherwise they will exist in both the *MCS* directory and the *Modules* subdirectory and modules may fail to load. Only place in the *Modules* subdirectory the assemblies of the modules and any other assemblies specific to your modules.

3.1 Configuring MCS to load distribution modules

To instruct the MCS's Distribution to load a distribution module, its configuration shall be updated. The configuration file is the XML file named *Distribution.exe.config* and located in the installation directory of the MCS. An XML element matching the following format must be added to the *distributionModules* section:

```
<add dataType="data-type-number" type="assembly-qualified-type-name" />
```

The *dataType* attribute is the number identifying the data type distributed by the module.

The *type* attribute is the assembly-qualified name[4] of the type implementing the *IDistributionModule* interface. Here is an example of the *distributionModules* section configured to load a distribution module:

```
<distributionModules>
  <add dataType="128"
       type="SwissCube.ImageService.DistributionModule, ImageServiceDistributionModule" />
</distributionModules>
```

Figure 5 - Example of *distributionModules* configuration section with a distribution module

[4] .NET Framework assembly-qualified type names: http://msdn.microsoft.com/en-us/library/system.type.assemblyqualifiedname.aspx

3.2 IDistributionModule interface

The *IDistributionModule* interface is defined in the *Mcs.Distribution.Contracts* assembly and resides in the *Epfl.SwissCube.GroundSegment.Mcs.Distribution* namespace. It is shown in Figure 6.

```
using System;
using System.ServiceModel;

namespace Epfl.SwissCube.GroundSegment.Mcs.Distribution
{
   // Defines the interface implemented by the distribution modules.
   public interface IDistributionModule
   {
      // Return the custom reporting data for the specified data type.
      //  spacecraftId: The Spacecraft ID for which data must be returned.
      //  dataType:     Data type of the custom reporting data requested.
      //  dataSubtype:  Data subtype of the custom reporting data requested.
      //  parameters:   Additional parameters of the custom reporting data request.
      // Returns: The custom reporting data requested.
      CustomReportingData GetCustomReportingData(
                  ushort spacecraftId,
                  byte dataType,
                  byte dataSubtype,
                  byte[] parameters);
   }
}
```

Figure 6 - *IDistributionModule* interface to be implemented by the distribution modules

This interface consists of only one method that takes the Spacecraft ID, data type and subtype as well as binary-serialized parameters whose meaning is specific to the module. These arguments precisely specify which CRD is requested. The distribution module must then return that data in the form of a *CustomReportingData* object. See chapter 1 *Custom Reporting Data* for details about the CRD type itself.

4 FIGURES AND TABLES

5 REFERENCES

[N1] ECSS-E-70-41A Ground systems and operations - Telemetry and telecommand packet utilization. 30 January 2003.

[N2] SC-ICD-1-0-MDC_Extensibility – Mission Data Client Extensibility User Guide