**Independent Validation of OEE Data in SFx**

The purpose of this document is to show the work that I have done with reading SFx data into a SQL Server database, running stored procedures on the data, and then independently verifying the results from SFx charts.

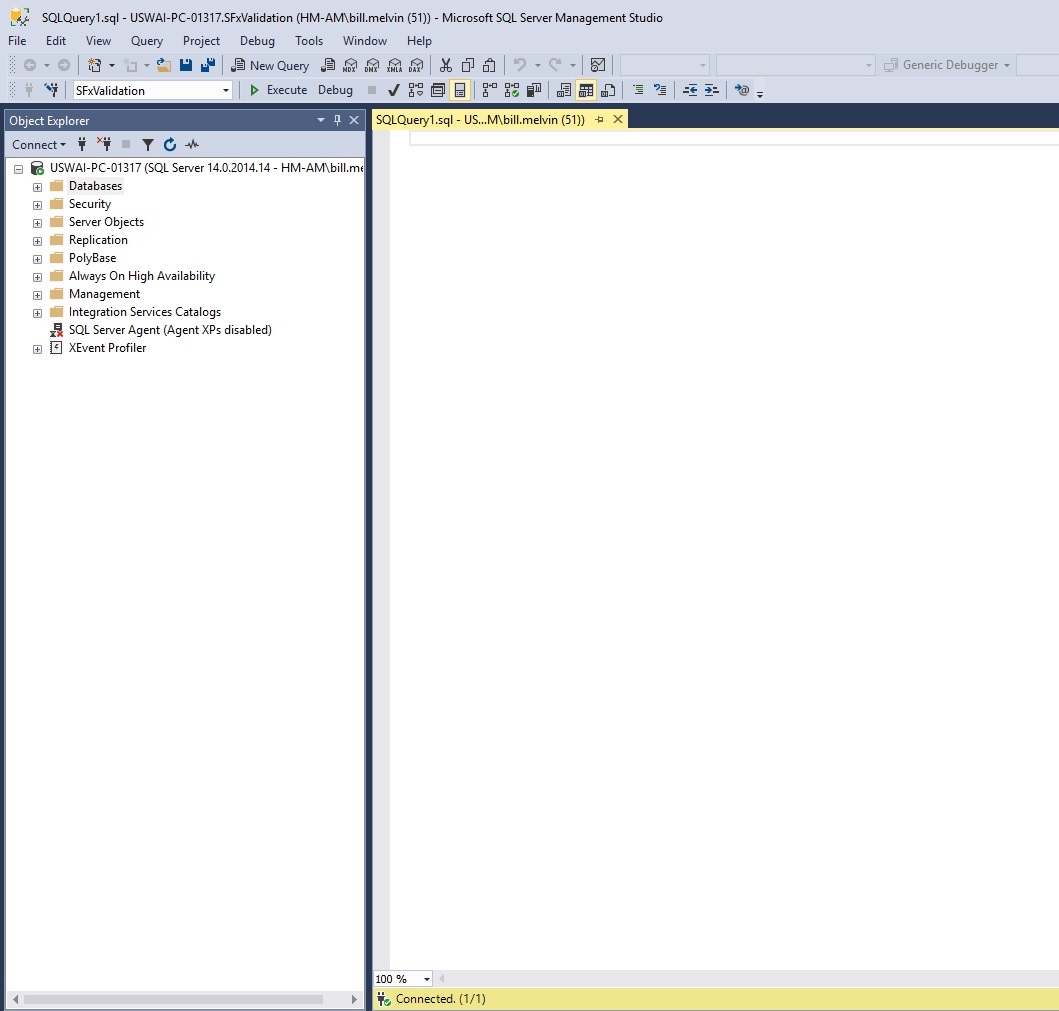
To help make the document more readable, I am putting code that can be executed in green.

**Prerequisites:**

1. SQL Server. For this exercise, even the free SQL Server Express version should work.
2. SQL Server Management Studio. This is included in some SQL Server distribution packages. In others, it must be downloaded separately. It is often referred to as SQL Server Tools.
3. A SQL Server account with enough privileges to create and run stored procedures. For most environments, the default Windows Authentication account should be sufficient.
4. SFxDataManager. This valuable tool developed by Barry Hockaday is the only way we currently have to bring the data into SQL Server.

**Executing commands in SQL Server Management Studio:**

When SQL Server Management Studio launches for the first time, no windows will appear. Clicking the “New Query” button will create a window for running commands. You will likely choose to have many of these windows open at the same time.



All of the commands given in this guide will be executed in one of these windows. If nothing is highlighted, then all of the commands will execute. If one/more lines are highlighted, then only those commands will be executed when the “Execute” button is pressed.

**Creating a Database:**

A database will be needed to store the tables and stored procedures referenced in this guide. In the previous screenshot, notice that SFxValidation appears above Object Explorer. This is the name of the active database for the window that is shown. I do not recommend running everything from the default “master” database as this will mix tables with system tables and make it harder to find your data. The selection box will easily let you change the active database.

You will probably only need to create a new database once. The simplest way:

1. Right click on Database (under Object Explorer) and select “New Database.”
2. Give the database a name (such as SFxValidation).
3. Click ok.

You will now have a database in the default location for SQL Server. This can now be selected in the selection box and appears when you expand the Databases.

**Creating the Tables:**

Several tables are used by the Stored Procedures. These are defined here. Make sure that the correct database is selected and run the following script:

create table telemetry(

[Start\_Time] datetime,

[End\_Time] datetime,

[Status] nvarchar(100));

create table stats(

[ES\_Status\_Record\_Count] int,

[Processed\_Status\_Record\_Count] int,

[Time\_To\_Process\_Status\_Data] float);

create table query\_parameters(

[StartDateTime] datetime,

[EndDateTime] datetime,

[UseSchedule] varchar(1));

create table runtime\_data(

[Runtime] float,

[NonRuntime] float,

[Unknown] float,

[SpecificRuntime] float,

[SpecificNonRuntime] float,

[SpecificUnknown] float,

[ScheduledTime] float);

create table status\_aggregation(

[IdleTime] float,

[BusyTime] float,

[ErrorTime] float,

[UnknownTime] float);

**Populating the query\_parameters table:**

The query\_parameters table tells the stored procedures the date range to use for the calculations. It also instructs whether or not to use schedule data (not implemented yet). This table should have one row with the date range. It can be added manually by right clicking the query\_parameters table and selecting “Edit Top 200 Rows” or by running a script like:

INSERT INTO query\_parameters (StartDateTime, EndDateTime, UseSchedule)

VALUES ('2019-05-18 13:58:45.650', '2019-05-19 08:59:26.930', 'N');

The values in this table should be changed to reflect the date range for the data to be retrieved.

**Creating the Stored Procedures:**

The Stored Procedures only need to be created once unless they are modified. If modified, they will need to be dropped and re-added. Here are the scripts for the current three stored procedures. Put each of these into their own window, click Execute, and they will be created.

These three stored procedures duplicate functionality from functions with the same names within the Platform solution. They are intended to provide an independent way to verify the data.

**ProcessStatusData**

Create procedure ProcessStatusData as

DECLARE @Status nvarchar(100);

DECLARE @TimeStampDate datetime;

DECLARE @LocalTimestampDate datetime;

DECLARE @CurrentStatus nvarchar(100);

DECLARE @LastTimeStamp datetime;

DECLARE @LastKnownTimestamp datetime;

DECLARE @StartTime datetime;

DECLARE @EndTime datetime;

DECLARE @UseSchedule nvarchar(1);

DECLARE @BeginExecutionTime datetime;

DECLARE @EndExecutionTime datetime;

DECLARE @TelemetryRecordsProcessed int;

DECLARE @TelemetryRecordsStored int;

DECLARE @ExecutionTime float;

DECLARE @SQL\_String nvarchar(1000);

SET @SQL\_String = 'delete from telemetry';

execute (@SQL\_String);

SET @SQL\_String = 'delete from stats';

execute (@SQL\_String);

SET @BeginExecutionTime = getdate();

DECLARE ParameterCursor CURSOR FOR

SELECT StartDateTime, EndDateTime, UseSchedule from query\_parameters;

OPEN ParameterCursor

FETCH NEXT FROM ParameterCursor INTO @StartTime, @EndTime, @UseSchedule

DECLARE TelemetryCursor CURSOR FOR

SELECT ChartStatus, TimestampDate, LocalTimestampDate from Pulse\_Status

where TimestampDate >= @StartTime and TimestampDate <= @EndTime order by TimestampDate asc;

OPEN TelemetryCursor

FETCH NEXT FROM TelemetryCursor INTO @Status, @TimeStampDate, @LocalTimestampDate

Set @LastKnownTimestamp = @TimestampDate;

PRINT @TimestampDate;

Set @CurrentStatus = ' ';

WHILE @@FETCH\_STATUS = 0

BEGIN

if @Status <> @CurrentStatus

BEGIN

if @CurrentStatus <> ' '

BEGIN

INSERT INTO TELEMETRY (Start\_Time, End\_Time, Status)

VALUES (@LastTimeStamp, @TimestampDate, @CurrentStatus);

Set @CurrentStatus = @Status;

Set @LastTimeStamp = @TimestampDate;

PRINT @Status + ' ' + RTRIM(CAST(@TimestampDate AS nvarchar(30)));

END;

else

BEGIN

Set @CurrentStatus = @Status;

Set @LastTimeStamp = @TimestampDate;

if @LastTimeStamp > @StartTime

BEGIN

INSERT INTO TELEMETRY (Start\_Time, End\_Time, Status)

VALUES (@StartTime, @LastTimeStamp, 'Unknown');

END;

END;

END

FETCH NEXT FROM TelemetryCursor INTO @Status, @TimeStampDate, @LocalTimestampDate

PRINT @TimestampDate;

Set @LastKnownTimestamp = @TimestampDate;

END

INSERT INTO TELEMETRY (Start\_Time, End\_Time, Status)

VALUES (@LastTimeStamp, @EndTime, @Status);

SET @EndExecutionTime = getdate();

set @ExecutionTime = datediff(ms, @BeginExecutionTime, @EndExecutionTime) / 1000.0;

DECLARE TelemetryCountCursor CURSOR FOR

Select Count(\*) from TELEMETRY;

OPEN TelemetryCountCursor

FETCH NEXT FROM TelemetryCountCursor INTO @TelemetryRecordsStored

DECLARE RecordsProcessedCursor CURSOR FOR

Select Count(\*) from Pulse\_Status

where TimestampDate >= @StartTime and TimestampDate <= @EndTime;

OPEN RecordsProcessedCursor

FETCH NEXT FROM RecordsProcessedCursor INTO @TelemetryRecordsProcessed

INSERT INTO STATS (ES\_Status\_Record\_Count, Processed\_Status\_Record\_Count, Time\_To\_Process\_Status\_Data)

Values (@TelemetryRecordsProcessed, @TelemetryRecordsStored, @ExecutionTime);

CLOSE TelemetryCursor

DEALLOCATE TelemetryCursor

CLOSE ParameterCursor

DEALLOCATE ParameterCursor

CLOSE TelemetryCountCursor

DEALLOCATE TelemetryCountCursor

CLOSE RecordsProcessedCursor

DEALLOCATE RecordsProcessedCursor

GO

**GetRuntimeData**

Create procedure GetRuntimeData as

DECLARE @DataStartTime datetime;

DECLARE @DataEndTime datetime;

DECLARE @UseSchedule nvarchar(1);

DECLARE @SQL\_String nvarchar(1000);

DECLARE @Runtime float;

DECLARE @NonRuntime float;

DECLARE @Unknown float;

DECLARE @SpecificRuntime float;

DECLARE @SpecificNonRuntime float;

DECLARE @SpecificUnknown float;

DECLARE @ScheduledTime float;

DECLARE @StartTime datetime;

DECLARE @EndTime datetime;

DECLARE @Status nvarchar(100);

DECLARE @TotalTime float;

SET @SQL\_String = 'delete from runtime\_data';

execute (@SQL\_String);

DECLARE ParameterCursor CURSOR FOR

SELECT StartDateTime, EndDateTime, UseSchedule from query\_parameters;

OPEN ParameterCursor

FETCH NEXT FROM ParameterCursor INTO @DataStartTime, @DataEndTime, @UseSchedule

DECLARE DataCursor CURSOR FOR

SELECT Start\_Time, End\_Time, Status from telemetry;

OPEN DataCursor

FETCH NEXT FROM DataCursor INTO @StartTime, @EndTime, @Status

Set @SpecificRuntime = 0.0;

Set @SpecificNonRuntime = 0.0;

Set @SpecificUnknown = 0.0;

WHILE @@FETCH\_STATUS = 0

BEGIN

If @Status = 'Busy'

BEGIN

Set @SpecificRuntime = @SpecificRuntime + datediff(ms, @StartTime, @EndTime) / 1000.0;

END

Else If @Status = 'Idle' or @Status = 'Error'

BEGIN

Set @SpecificNonRuntime = @SpecificNonRuntime + datediff(ms, @StartTime, @EndTime) / 1000.0;

END

Else

BEGIN

Set @SpecificUnknown = @SpecificUnknown + datediff(ms, @StartTime, @EndTime) / 1000.0;

END

FETCH NEXT FROM DataCursor INTO @StartTime, @EndTime, @Status

END

Set @TotalTime = @SpecificRuntime + @SpecificNonRuntime + @SpecificUnknown;

if @TotalTime > 0

BEGIN

Set @Runtime = @SpecificRuntime / @TotalTime;

Set @NonRuntime = @SpecificNonRuntime / @TotalTime;

Set @Unknown = @SpecificUnknown / @TotalTime;

Set @ScheduledTime = @TotalTime;

END

INSERT INTO RUNTIME\_DATA (Runtime, NonRuntime, Unknown, SpecificRuntime, SpecificNonRuntime, SpecificUnknown, ScheduledTime)

Values (@Runtime, @NonRuntime, @Unknown, @SpecificRuntime, @SpecificNonRuntime, @SpecificUnknown, @ScheduledTime);

CLOSE ParameterCursor

DEALLOCATE ParameterCursor

CLOSE DataCursor

DEALLOCATE DataCursor

GO

**GetStatusAggregation**

Create procedure GetStatusAggregation as

DECLARE @DataStartTime datetime;

DECLARE @DataEndTime datetime;

DECLARE @UseSchedule nvarchar(1);

DECLARE @SQL\_String nvarchar(1000);

DECLARE @IdleTime float;

DECLARE @BusyTime float;

DECLARE @ErrorTime float;

DECLARE @UnknownTime float;

DECLARE @StartTime datetime;

DECLARE @EndTime datetime;

DECLARE @Status nvarchar(100);

SET @SQL\_String = 'delete from status\_aggregation';

execute (@SQL\_String);

DECLARE ParameterCursor CURSOR FOR

SELECT StartDateTime, EndDateTime, UseSchedule from query\_parameters;

OPEN ParameterCursor

FETCH NEXT FROM ParameterCursor INTO @DataStartTime, @DataEndTime, @UseSchedule

DECLARE DataCursor CURSOR FOR

SELECT Start\_Time, End\_Time, Status from telemetry;

OPEN DataCursor

FETCH NEXT FROM DataCursor INTO @StartTime, @EndTime, @Status

Set @IdleTime = 0.0;

Set @BusyTime = 0.0;

Set @ErrorTime = 0.0;

Set @UnknownTime = 0.0;

WHILE @@FETCH\_STATUS = 0

BEGIN

If @Status = 'Busy'

BEGIN

Set @BusyTime = @BusyTime + datediff(ms, @StartTime, @EndTime) / 1000.0;

END

Else If @Status = 'Idle'

BEGIN

Set @IdleTime = @IdleTime + datediff(ms, @StartTime, @EndTime) / 1000.0;

END

Else If @Status = 'Error'

BEGIN

Set @ErrorTime = @ErrorTime + datediff(ms, @StartTime, @EndTime) / 1000.0;

END

Else

BEGIN

Set @UnknownTime = @UnknownTime + datediff(ms, @StartTime, @EndTime) / 1000.0;

END

FETCH NEXT FROM DataCursor INTO @StartTime, @EndTime, @Status

END

INSERT INTO STATUS\_AGGREGATION (IdleTime, BusyTime, ErrorTime, UnknownTime)

Values (@IdleTime, @BusyTime, @ErrorTime, @UnknownTime);

CLOSE ParameterCursor

DEALLOCATE ParameterCursor

CLOSE DataCursor

DEALLOCATE DataCursor

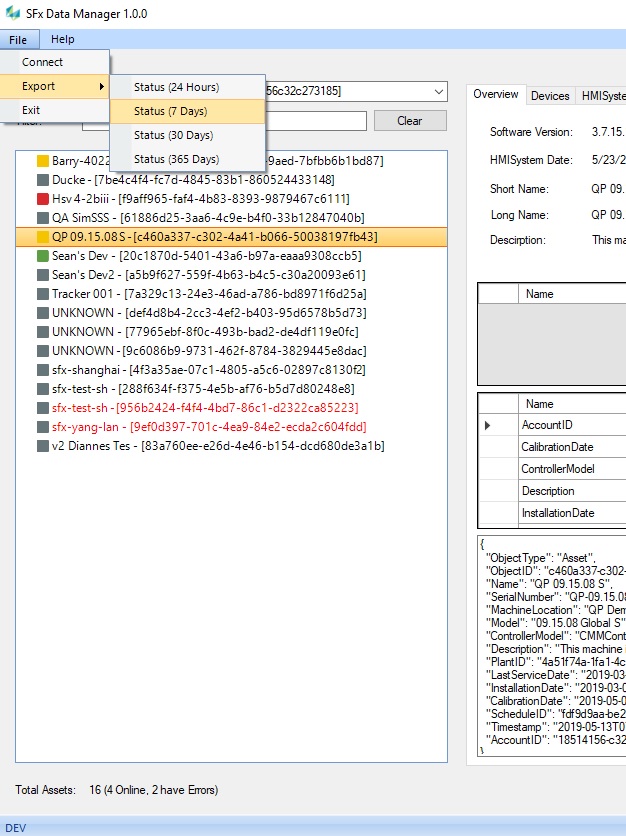
GO

**Running the verification test:**

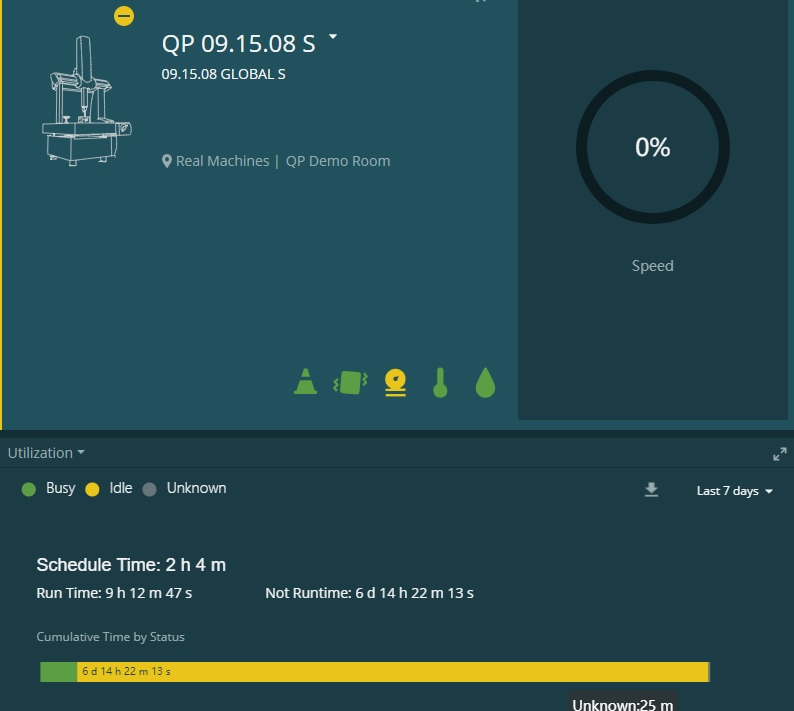
Now that we have SQL Server setup, the tables created, and the stored procedures in place, we are ready to go through the steps for verifying data. The rest of the document will be a walk-through for the steps that will be needed for each run. The example we are going to do is 7 days, but we could just as easily do the 24 hour or 30 day tests.

Note that the stored procedures automatically delete existing data in their tables when executed. This will save you the step of clearing out these tables. The query\_parameters table does need to be updated for the appropriate date range.

1. Connect to the VPN. These instructions are referenced from the SFx Data Manager tool and are subject to change.
2. Launch the SFx Data Manager tool.
3. Connect to an instance. In this example, we are connecting to DEV.
4. Once the SFx Data Manager tool has loaded, we will see all of the data that is available. In this case, we want the data for “QP 09.15.08”.
5. Highlight the desired data, select Export from the File menu, and then the time interval desired (7 days in this case). This will retrieve in JSON format the last 7 days of data for this device.



1. Open up SFx, navigate to the asset, set the time interval, and take a screenshot. This is data that will be compared against.



1. Now we have the JSON data and a screenshot that were from nearly the same time. I have not come up with a way to synchronize the data more exactly. I suggest doing these steps as quickly as possible to try to make the results as close as possible.
2. The JSON file is not in the correct format to be read into the database. Here are the steps to modify the file:
   1. Open the JSON file created in step 5 with Notepad.
   2. Delete everything before the first brace “{“.
   3. Insert a “[“ character before the first brace.
   4. Select Replace under the Edit menu.
   5. Find “}” and replace with “},”. Select Replace All.
   6. Scroll down to the bottom of the file. Delete the final comma “,”.
   7. Insert a “]” character after the last brace.
   8. Click Save. The data is now formatted to be read into SQL Server.
3. If you have run this script before, the Pulse\_Status table will already exist. It will need to be deleted. The easiest way to do this is to right click on the table and select Delete.
4. Within SQL Server Managent Studio, execute this script in its own window. Change the name of the JSON file to refer to the path and name of this file.

DECLARE @JSON VARCHAR(MAX)

SELECT @JSON = BulkColumn

FROM OPENROWSET

(BULK 'D:\text\c460a337-c302-4a41-b066-50038197fb43-7-Days.json', SINGLE\_CLOB)

AS j

SELECT \* into Pulse\_Status

from OPENJSON (@JSON)

WITH (CrashStatus nvarchar(100), VibrationStatus nvarchar(100), AirPressureStatus nvarchar(100),

HumidityStatus nvarchar(100), IlluminanceStatus nvarchar(100), TemperatureStatus nvarchar(100),

MessageType nvarchar(100), Status nvarchar(100), DeviceID nvarchar(100),

IoTDeviceID nvarchar(100), Timestamp nvarchar(100), LocalTimestamp nvarchar(100),

DeviceClass nvarchar(100), DeviceType nvarchar(100))

alter table Pulse\_Status add TimestampDate datetimeoffset;

alter table Pulse\_Status add LocalTimestampDate datetimeoffset;

alter table Pulse\_Status add ChartStatus nvarchar(100);

GO

update Pulse\_Status set TimestampDate = Parse(Timestamp as datetimeoffset);

update Pulse\_Status set LocalTimestampDate = Parse(LocalTimestamp as datetimeoffset);

update Pulse\_Status set ChartStatus = 'Idle'

where Status in ('Idle', 'IdleWarning', 'IdleCritical', 'Creating', 'CreatingWarning',

'CreatingCritical');

update Pulse\_Status set ChartStatus = 'Busy' where Status in ('Running', 'RunningWarning',

'RunningCritical');

update Pulse\_Status set ChartStatus = 'Waiting' where Status in ('Waiting', 'WaitingWarning',

'WaitingCritical');

update Pulse\_Status set ChartStatus = 'Unknown' where ChartStatus is null;

1. Now is a good time to update the query\_parameters table with the date range. One way to find a good date range is to bring up Kibana and see the date range given by one of the APIs. For example:



1. In our case, I know that the data was pulled near the top of the hour. So, my 7 day date range will look like this:



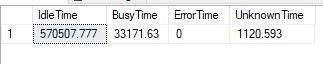
1. Now that the data is in the database, you are ready to run the stored procedures:

Execute ProcessStatusData;

Execute GetRuntimeData;

Execute GetStatusAggregation;

1. Now the data has been read into the database and the stored procedures have given us our results. For this example, we can compare the results in the status\_aggregation table with the screen shot from SFx we took earlier.



1. An obvious improvement would be to modify the scripts to convert these times (in seconds) into days, hours, minutes, and seconds. As it stands, we will need to do that manually. This results in:

Idle Time: 570507 seconds = 6 days, 14 hours, 28 minutes, 27 seconds

Busy Time: 33171 seconds = 9 hours, 12 minutes, 51 seconds

Unknown Time: 1120 seconds = 18 minutes, 40 seconds

From the earlier screenshot, the values we had were:

Idle Time: 6days, 14 hours, 22 minutes

Busy Time: 9 hours, 12 minutes, 51 seconds

Unknown Time: 25 minutes

The Busy time is an exact match. Only a discrepancy in the Idle Time and Unknown Time gives the difference. It is possible that the schedule accounts for this difference. Or, a bug in the SQL Server script or in the API. Only a detailed analysis will identify the root cause of this discrepancy.

This completes the demonstration of the code for independent OEE verification. This approach can be extended to cover data from all charts within SFx. Improvements can be made to make the process easier to run.