

ACSIL Interface - Custom Chart Bars

Related Documentation

- [ACSIL Interface Members - Introduction](#)
 - [ACSIL Interface Members - Variables and Arrays](#)
 - [ACSIL Interface Members - sc.Input Array](#)
 - [ACSIL Interface Members - sc.Subgraph Array](#)
 - [ACSIL Interface Members - Functions](#)
-

On This Page

- [Introduction](#)
 - [ACSIL Interface Members for Custom Chart Bars](#)
 - [s_CustomChartBarInterface Interface Members for Custom Bar Function](#)
 - [Detecting when Chart Needs to be Reloaded](#)
 - [Complete Code Example](#)
-

Introduction

It is supported to create custom chart bars using the [Advanced Custom Study Interface and Language](#). You will create a custom chart bars study. So it is a study which then creates custom chart bars.

These custom chart bars are implemented in the same way as the different **Bar Period Types** for Intraday charts in **Chart >> Chart Settings >> Intraday Chart Bar Period** are implemented.

To programmatically determine if custom chart bars are being used on a chart, use the function [sc.GetBarPeriodParameters](#). The **s_BarPeriod::ACSILCustomChartStudyName** string variable will contain the name of the custom chart bars.

When a instance of the study is added to the chart the [Hide Study](#) setting is automatically enabled. It is not possible to disable this.

ACSIL Interface Members for Custom Chart Bars

sc.UsesCustomChartBarFunction

Type: Read/write integer variable.

sc.UsesCustomChartBarFunction when set to 1 or a nonzero value, specifies that the study

function is using a custom bar building function which will set the custom study to become a custom chart bar study. When this is set to 0, the study function is a normal study that does not build custom chart bars.

For complete working example, refer to [Complete Code Example](#).

Example

```
if (sc.SetDefaults)
{
    sc.UsesCustomChartBarFunction = 1;
    sc.fp_ACSCustomChartBarFunction = CustomChartBarBuildingFunction;
}
```

sc.fp_ACSCustomChartBarFunction

Type: Read/write integer variable.

sc.fp_ACSCustomChartBarFunction needs to be set to the custom study function which builds the custom chart bars for the study. This function is called from a separate thread.

For complete working example, refer to [Complete Code Example](#).

Example

```
if (sc.SetDefaults)
{
    sc.UsesCustomChartBarFunction = 1;
    sc.fp_ACSCustomChartBarFunction = CustomChartBarBuildingFunction;
}
```

s_CustomChartBarInterface Interface Members for Custom Bar Function

All of the following are Input members of the **ChartBarInterface** structure of type **s_CustomChartBarInterface**, for use by the custom bar building function to access the underlying chart data for building chart bars.

The following is an example declaration of this function which receives a reference to the **ChartBarInterface** structure.

SCSFExport CustomRangeChartBarBuildingFunction(SCCustomChartBarInterfaceRef ChartBarInterface).

- **ChartDataEnum ChartDataType:** Indicates if this chart is based on historical Daily data or Intraday data. Possible values: **DAILY_DATA** = 1, **INTRADAY_DATA** = 2.
- **unsigned char IsDeterminingIfShouldStartNewBar:** Set to 1 if the custom bar

building function needs to indicate if should start a new chart bar. Otherwise, it is 0.

- **unsigned char IsFinalProcessingAfterNewOrCurrentBar:** Set to 1 if the custom bar building function is being called after data has been added to a chart bar, whether it is a new chart bar or not. Otherwise, it is 0.
- **unsigned char IsInsertFileRecordsProcessing:** When the custom bar building function is called and this flag is true when there is a new intraday file record read and it has not yet been processed. That new intraday file record is not provided, but this is an opportunity to perform a "split" operation on the most recent chart bar and break it up into smaller records by reducing its size and putting the excess into `ChartBarInterface.NewRecordToInsert`.

Set `ChartBarInterface.InsertNewRecord` to 1 if `ChartBarInterface.NewRecordToInsert` has been filled out with a new record. In this case, it will be processed normally which means that this function will be called first with `ChartBarInterface.IsDeterminingIfShouldStartNewBar` set to 1, and again with `ChartBarInterface.IsFinalProcessingAfterNewOrCurrentBar` set to 1.

When the normal processing is done, this function will be called again with `ChartBarInterface.IsInsertFileRecordsProcessing` set to 1. If there is no further processing to do, then just return without doing anything further.

Generally the purpose of `ChartBarInterface.IsInsertFileRecordsProcessing` is to split bars which have been created from Intraday data records that are greater than 1 tick and also to perform splitting for Renko and Range bars when there are price gaps.

Splitting is a multi-iteration process. It is not done all at once. For example, if there is an existing chart bar with a range of 10, and it needs to be broken up into 9 additional bars with a range of 1 each, then the first step is to set its range to 1 and insert a new file record through **`ChartBarInterface.NewRecordToInsert`** with a range of 9.

When the standard record processing is complete, this function is called again with `ChartBarInterface.IsInsertFileRecordsProcessing` set to 1, and this process continues by putting the excess range, 8, back into `ChartBarInterface.NewRecordToInsert` until the process is complete.

Internally Sierra Chart is going to divide up the volume among each chart bar evenly during the splitting process.

- **s_IntradayRecord NewFileRecord:** This is the currently being processed Intraday data file record.
- **uint32_t CurrentBarIndex:** The zero-based array index of the current chart bar being built.
- **uint32_t ValueFormat:** The [Price Display Format](#) of the chart.
- **unsigned char IsNewFileRecord:** A variable indicating whether **NewFileRecord** is a new record during the bar building process. 1 is true. 0 is false.
- **unsigned char BarHasBeenCutAtStartOfSession:** A variable indicating whether the current chart bar has just been started because of the start of the trading session based

on the Session Times of the chart. 1 is true. 0 is false.

- **unsigned char IsNewChartBar**: A variable indicating whether the current chart bar is a newly started chart bar. 1 is true. 0 is false.
- **unsigned char IsFirstBarOfChart**: A variable indicating if the current bar is the first bar of the chart. 1 is true. 0 is false.
- **float TickSize**: The minimum price increment the chart is set to.
- **float BidPrice**: The bid price associated with the most recent trade in the **NewFileRecord** data structure.
- **float AskPrice**: The ask price associated with the most recent trade in the **NewFileRecord** data structure.
- **c_VAPContainer VolumeAtPriceForBars: VolumeAtPriceForBars** is a pointer to a **c_VAPContainer** object, which is a high-performance data container containing the volume at price data for the chart bar. For further details, refer to [sc.VolumeAtPriceForBars](#). This has been added in version 1793.

The following are the **s_CustomChartBarInterface** Output members, meaning they are set by the custom bar building function.

- **unsigned char StartNewBarFlag**: Set to 1 to start a new chart bar. In this case **NewFileRecord** becomes part of the new chart bar. A new bar will also be started if [Chart >> Chart Settings >> New Bar at Session Start](#) is enabled, and according to the conditions as defined for that option.
- **unsigned char InsertNewRecord**: This variable is used with **IsInsertFileRecordsProcessing**.
- **s_IntradayRecord NewRecordToInsert**: This variable is used with **IsInsertFileRecordsProcessing**.

s_CustomChartBarInterface Functions

- **float& GetChartBarValue(int SubgraphIndex, int BarIndex)**: For accessing the bar values at each bar index. The returned value is a reference and is read/write.
- **const SCDateTime& GetChartBarDateTime(int BarIndex)**: For accessing the Date-Time of the chart bar at the specified bar index. The returned value is a constant reference. To modify this Date-Time you need to perform a const cast to remove the constant attribute. We make no guarantee whether modifying the Date-Time would cause a problem or not. At the time of this writing it is believed to not cause an issue.
- **SCInputRef& GetInput(int InputIndex)**: This function is used to return a reference to the study [Input](#) at the specified InputIndex.
- **int& GetPersistentInt(int Key)**: Refer to [sc.GetPersistentInt](#).
- **float& GetPersistentFloat(int Key)**: Refer to [sc.GetPersistentInt](#).
- **double& GetPersistentDouble(int Key)**: Refer to [sc.GetPersistentInt](#).
- **int64_t& GetPersistentInt64(int Key)**: Refer to [sc.GetPersistentInt](#).
- **SCDateTime& GetPersistentSCDateTime(int Key)**: Refer to [sc.GetPersistentInt](#).

Additional Input members

- **float BidPrice**: The bid price associated with the last trade in NewFileRecord.
- **float AskPrice**: The ask price associated with the last trade in NewFileRecord.

Detecting when Chart Needs to be Reloaded

Complete Code Example

The following series of functions are a complete example for building custom Range bars. The Range bar building is more simplified than the actual Range bar construction within Sierra Chart.

Example

```

/*=====
Returns 1 for an up bar, and -1 for a down bar. If the direction can not
be determined within the past 10 bars, this function will return 1 (up)
as the direction.
-----*/
int GetRangeBarDirection(SCCustomChartBarInterfaceRef ChartBarInterface, int BarIndex)
{
    int LastIndex = BarIndex;
    int NumIterations = 0;
    while (NumIterations < 10)
    {
        float& BarOpen = ChartBarInterface.GetChartBarValue(SC_OPEN, LastIndex);
        float& BarLast = ChartBarInterface.GetChartBarValue(SC_LAST, LastIndex);

        // If the close is below the open, then the bar is down.
        if (BarLast < BarOpen)
        {
            // Down
            return -1;
        }

        // If the close is above the open, then the bar is up.
        if (BarLast > BarOpen)
        {
            // Up
            return 1;
        }

        // If the close is the same as the open, then we need to look at the
        // difference between the current close and the prior close.

        if (LastIndex < 1)
            break;

        float& PriorBarLast = ChartBarInterface.GetChartBarValue(SC_LAST, LastIndex - 1);

        // If the current close is below the prior close, then the bar is
        // down.
        if ( BarLast < PriorBarLast)
        {
            // Down
            return -1;
        }

        // If the current close is above the prior close, then the bar is up.
        if ( BarLast > PriorBarLast )
        {
            // Up
            return 1;
        }
    }
}

```

```

    }

    // If the current close is the same as the prior close, repeat the
    // algorithm using the previous bar.
    --LastIndex;
    ++NumIterations;
}

// Direction could not be determined, so always use up.
return 1;
}
/*=====
-----*/
SCSFExport CustomRangeChartBarBuildingFunction(SCCustomChartBarInterfaceRef ChartBarInterface)
{
    ChartBarInterface.StartNewBarFlag = 0;

    SCInputRef RangeInput = ChartBarInterface.GetInput(0);
    float NeededRangeAsFloat = RangeInput.GetInt() * ChartBarInterface.TickSize;

    if (ChartBarInterface.IsDeterminingIfShouldStartNewBar)
    {
        if (ChartBarInterface.BarHasBeenCutAtStartOfSession)
        {
            // This will be true when 'Chart >> Chart Settings >> New Bar at Session Start' is enabled and that condition h
        }

        int Direction = GetRangeBarDirection(ChartBarInterface, ChartBarInterface.CurrentBarIndex);

        float BarHigh = ChartBarInterface.GetChartBarValue(SC_HIGH, ChartBarInterface.CurrentBarIndex);
        float BarLow = ChartBarInterface.GetChartBarValue(SC_LOW, ChartBarInterface.CurrentBarIndex);

        if (Direction == 1)
        {
            BarHigh = max(BarHigh, ChartBarInterface.NewFileRecord.GetHigh());

            float Range = BarHigh - BarLow;

            if (Range > NeededRangeAsFloat)
                ChartBarInterface.StartNewBarFlag = 1;
        }
        else if (Direction == -1)
        {
            BarLow = min(BarLow, ChartBarInterface.NewFileRecord.GetLow());

            float Range = BarHigh - BarLow;

            if (Range > NeededRangeAsFloat)
                ChartBarInterface.StartNewBarFlag = 1;
        }
    }
    else if (ChartBarInterface.IsFinalProcessingAfterNewOrCurrentBar)
    {
        if (ChartBarInterface.IsNewChartBar && ChartBarInterface.CurrentBarIndex > 0)
        {
            int Direction = GetRangeBarDirection(ChartBarInterface, ChartBarInterface.CurrentBarIndex - 1);

            if (Direction == 1)
            {
                float& r_BarHigh = ChartBarInterface.GetChartBarValue(SC_HIGH, ChartBarInterface.CurrentBarIndex - 1);
                float& r_BarLast = ChartBarInterface.GetChartBarValue(SC_LAST, ChartBarInterface.CurrentBarIndex - 1);

                r_BarLast = r_BarHigh;
            }
            else if (Direction == -1)

```

```

{
    float& r_BarLow = ChartBarInterface.GetChartBarValue(SC_LOW, ChartBarInterface.CurrentBarIndex - 1);
    float& r_BarLast = ChartBarInterface.GetChartBarValue(SC_LAST, ChartBarInterface.CurrentBarIndex - 1);

    r_BarLast = r_BarLow;
}
}
}
else if (ChartBarInterface.IsInsertFileRecordsProcessing)
{

    // If the range of the last bar is greater than the range setting when
    // there is a new record.

    int Direction = GetRangeBarDirection(ChartBarInterface, ChartBarInterface.CurrentBarIndex);

    float& r_BarHigh = ChartBarInterface.GetChartBarValue(SC_HIGH, ChartBarInterface.CurrentBarIndex);
    float& r_BarLow = ChartBarInterface.GetChartBarValue(SC_LOW, ChartBarInterface.CurrentBarIndex);
    float& r_BarOpen = ChartBarInterface.GetChartBarValue(SC_OPEN, ChartBarInterface.CurrentBarIndex);
    float& r_BarLast = ChartBarInterface.GetChartBarValue(SC_LAST, ChartBarInterface.CurrentBarIndex);

    float Range = r_BarHigh - r_BarLow;

    if (Range <= NeededRangeAsFloat)
    {
        ChartBarInterface.InsertNewRecord = 0;
        return;
    }

    // Create a new overflow record to take everything from the last bar that
    // does not fit in the range setting, depending on the direction of the
    // last bar.

    ChartBarInterface.InsertNewRecord = 1;

    if (Direction == 1) // Up
    {
        ChartBarInterface.NewRecordToInsert.High = r_BarHigh;
        ChartBarInterface.NewRecordToInsert.Close = r_BarLast;

        // Lower the high of the last bar so that the range of the last bar
        // is equal to the range setting.

        float NewHigh = r_BarLow + NeededRangeAsFloat;
        r_BarHigh = NewHigh;

        if (r_BarLast > r_BarHigh)
        {
            r_BarLast = r_BarHigh;
        }

        if (r_BarOpen > r_BarHigh)
        {
            r_BarOpen = r_BarHigh;
        }

        ChartBarInterface.NewRecordToInsert.Low = r_BarHigh;
        ChartBarInterface.NewRecordToInsert.Open = r_BarHigh + ChartBarInterface.TickSize;

        if (ChartBarInterface.NewRecordToInsert.Close < ChartBarInterface.NewRecordToInsert.Low)
            ChartBarInterface.NewRecordToInsert.Close = ChartBarInterface.NewRecordToInsert.Low;
    }
    else // Down
    {
        ChartBarInterface.NewRecordToInsert.Low = r_BarLow;
        ChartBarInterface.NewRecordToInsert.Close = r_BarLast;

        // Raise the low of the last bar so that the range of the last bar is
        // equal to the range setting.
    }
}

```

```

// equal to the range setting.

float NewLow = r_BarHigh - NeededRangeAsFloat;

r_BarLow = NewLow;

if (r_BarLast < r_BarLow)
{
    r_BarLast = r_BarLow;
}

if (r_BarOpen < r_BarLow)
{
    r_BarOpen = r_BarLow;
}

ChartBarInterface.NewRecordToInsert.High = r_BarLow;
ChartBarInterface.NewRecordToInsert.Open = r_BarLow - ChartBarInterface.TickSize;

if (ChartBarInterface.NewRecordToInsert.Close > ChartBarInterface.NewRecordToInsert.High)
    ChartBarInterface.NewRecordToInsert.Close = ChartBarInterface.NewRecordToInsert.High;
}
}
}

/*=====
-----*/
SCSFExport scsf_CustomRangeChartBarsExample(SCStudyInterfaceRef sc)
{
    SCInputRef Input_RangePerBar = sc.Input[0];

    if (sc.SetDefaults)
    {
        // Set the configuration and defaults
        sc.GraphRegion = 0;
        sc.GraphName = "Custom Range Chart Bars Example";
        sc.UsesCustomChartBarFunction = 1;

        // This function is called from another thread.
        sc.fp_ACSCustomChartBarFunction = CustomRangeChartBarBuildingFunction;

        sc.AutoLoop = 0; // Always use manual looping.

        //sc.MaintainAdditionalChartDataArrays = 1;
        //sc.AllocateAndNameRenkoChartBarArrays = 1;

        Input_RangePerBar.Name = "Range Per Bar in Ticks";
        Input_RangePerBar.SetInt(10);
        Input_RangePerBar.SetIntLimits(1, INT_MAX);

        return;
    }

    int& r_ChartDataReloadedFlag = sc.GetPersistentInt(CUSTOM_CHART_BAR_RELOAD_FLAG_KEY);

    if (r_ChartDataReloadedFlag == 0)
    {
        sc.FlagToReloadChartData = true;
        r_ChartDataReloadedFlag = 1;
    }

    // It is necessary for the study function which creates custom chart bars to detect changes with its own input se

    const int RangePerBar = Input_RangePerBar.GetInt();

    int& r_PriorRangePerBar = sc.GetPersistentInt(1);

```



```

if (r_PriorRangePerBar != 0 && r_PriorRangePerBar != RangePerBar)
    sc.FlagToReloadChartData = true;

r_PriorRangePerBar = RangePerBar;

// Set the graph name for this custom bar type.
if (sc.IsFullRecalculation)
    sc.GraphName.Format("Custom Range Bar = %d", RangePerBar);
}

/*=====
This function can take a well commented description of the relevant variables of the ChartBarInterface structure
-----*/

SCSFExport CustomChartBarBuildingFunction(SCCustomChartBarInterfaceRef ChartBarInterface)
{
    // This is the bar building function which is used along with
    // scsf_CustomChartBarsExample to build custom chart bars. This
    // function tells Sierra Chart when to start a new bar and allows
    // the custom study to make any necessary bar modifications and
    // insert its own file records.

    // It is Sierra Chart itself, which will combine the Intraday
    // data records into bars and set the Date-Time, Open, High,
    // Low, Last, Volume, Bid Volume, Ask Volume, and Volume at
    // Price data for the chart bars.

    // This is set to 0 by Sierra Chart upon each entry into this
    // function, but just setting it here for clarity.

    ChartBarInterface.StartNewBarFlag = 0;

    if (ChartBarInterface.IsDeterminingIfShouldStartNewBar)
    {
        // This flag is set to 1 and this function is called after
        // the initial processing of the intraday file record but
        // before it is added to the most recent chart bar. Set
        // ChartBarInterface.StartNewBarFlag to 1 to indicate to
        // start a new bar or 0 to indicate not to start a new
        // chart bar.

        // Below is an example of how a volume-based chart bar is
        // built. Once the volume is equal to or greater than the
        // volume specified through the study Input, then a new
        // chart bar is indicated to be started. After a new bar
        // is added, then ChartBarInterface.NewFileRecord is
        // added to that bar.

        if (ChartBarInterface.BarHasBeenCutAtStartOfSession)
        {
            // This will be true when 'Chart >> Chart Settings
            // >> New Bar at Session Start' is enabled and that
            // condition has been met.
        }

        SCInputRef VolumeInput = ChartBarInterface.GetInput(0);
        const uint64_t CurrentBarVolume = (uint64_t)ChartBarInterface.GetChartBarValue(SC_VOLUME, ChartBarInt

        if (CurrentBarVolume >= VolumeInput.GetInt())
        {
            ChartBarInterface.StartNewBarFlag = 1;

            return;
        }
    }
    else if (ChartBarInterface.IsFinalProcessingAfterNewOrCurrentBar)

```

```

else if (ChartBarInterface.IsFinalProcessingAfterNewOrCurrentBar)
{
    // This function is called and this flag is true after and
    // Intraday file record is fully processed. This gives the
    // custom study the ability to make any modifications to
    // the most recent chart bar. Or theoretically any chart
    // bar through the ChartBarInterface.GetChartBarValue()
    // function.

    //This will be true when the first bar is added to the chart
    if (ChartBarInterface.IsFirstBarOfChart)
    {
        //perform initialization.
    }

    // Example to set the last trade price of the bar to the high
    // of the bar after the bar is finished building.

    if (ChartBarInterface.IsNewChartBar)
    {
        if (ChartBarInterface.CurrentBarIndex >= 1)
        {
            ChartBarInterface.GetChartBarValue(SC_LAST, ChartBarInterface.CurrentBarIndex - 1) = ChartBarInterface
        }
    }

    return;
}
else if (ChartBarInterface.IsInsertFileRecordsProcessing)
{
}

return;
}
}

```

*Last modified Wednesday, 22nd February, 2023.