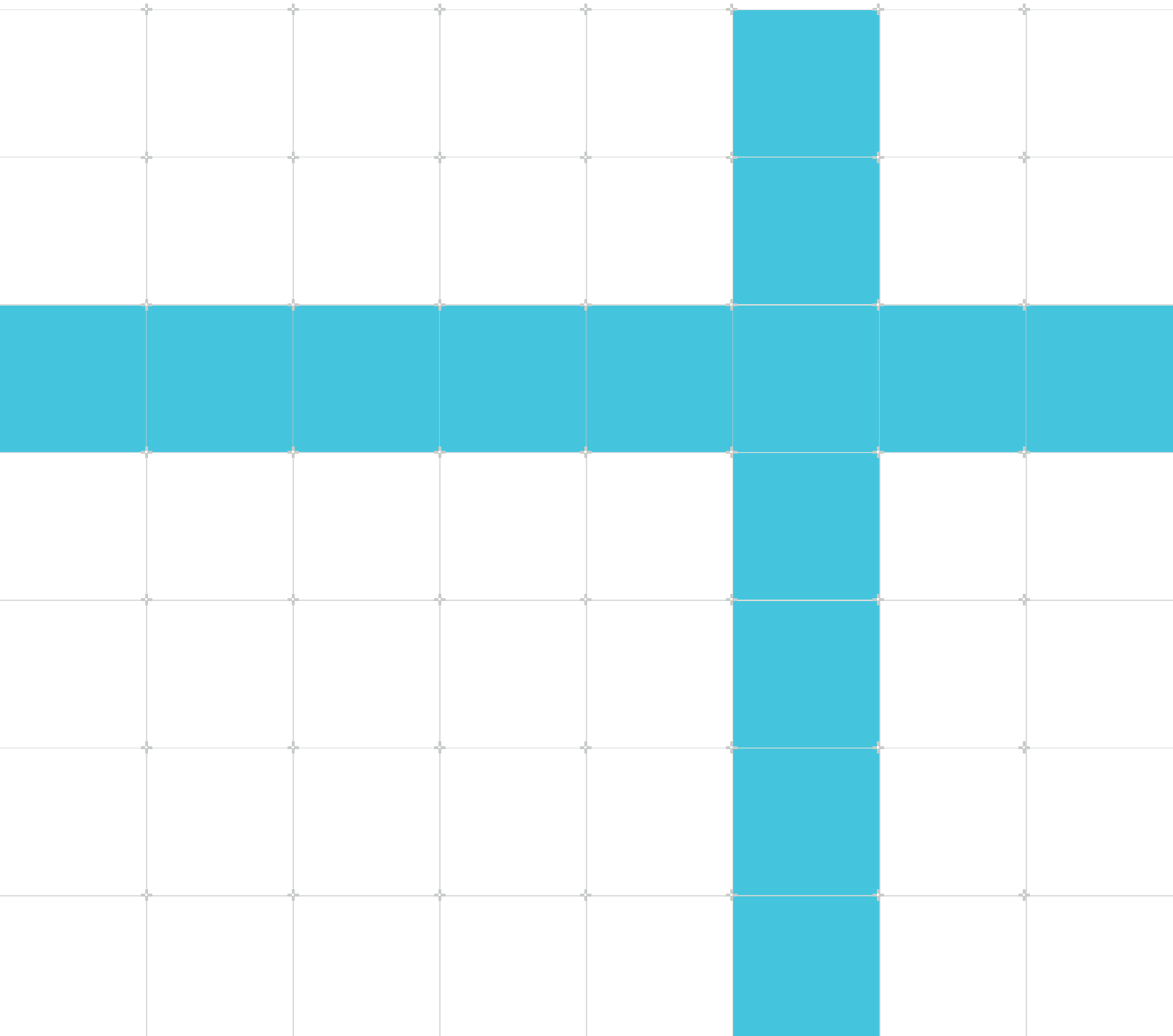
How to Create an ETR Configuration Tab in the DS-5 DTSL Options Dialog

Version 1.0

Issue 00
102663_0100_00_en



How to Create an ETR Configuration Tab in the DS-5 DTSL Options Dialog

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	4 November 2019	Non-Confidential	First release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

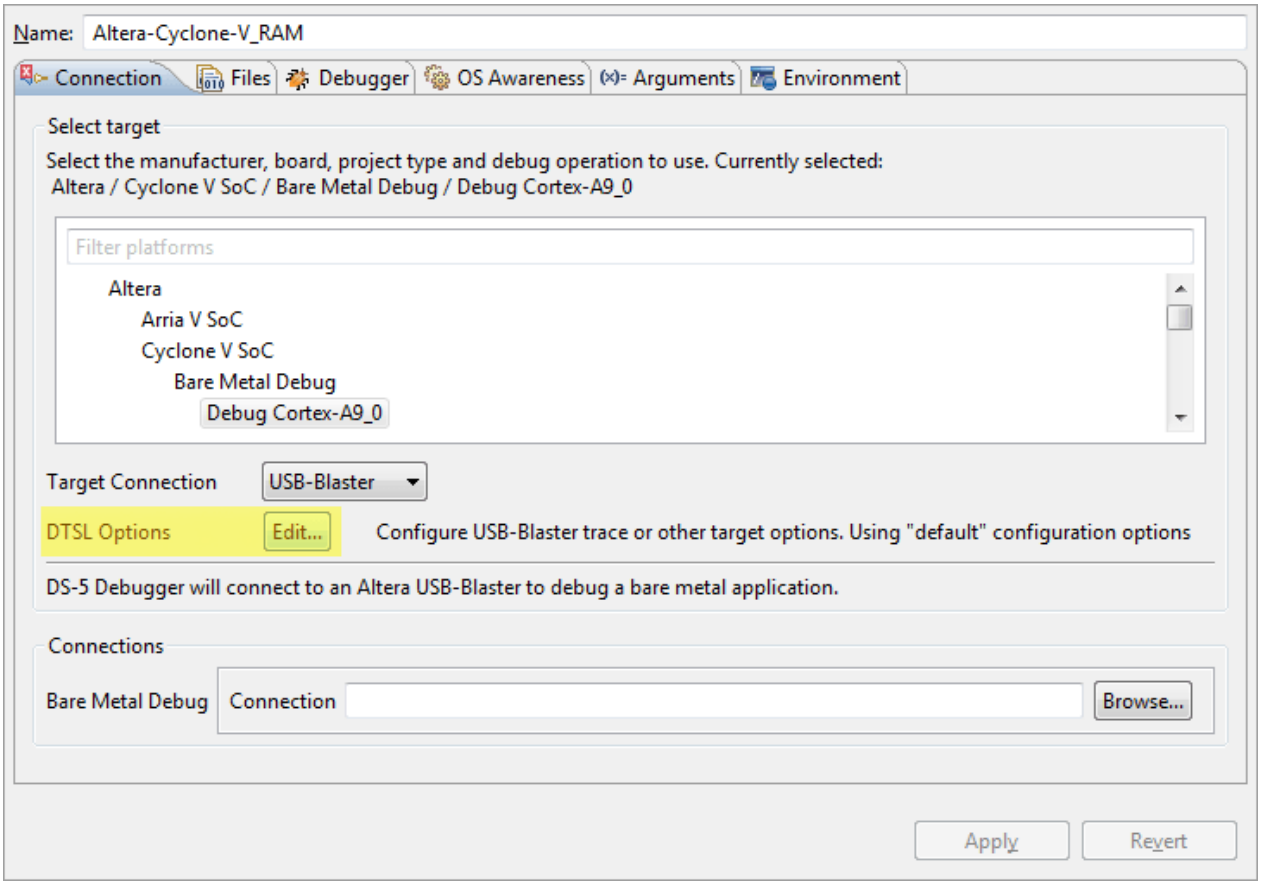
Learn how to create and customize an Embedded Trace Router (ETR) tab in the DTSL options dialog in DS-5 Development Studio using a simple Python script.

The controls in the DTSL Options dialog enable you to customize a debug session - configuring debug and trace according to the needs of a particular debug session. The controls and their supporting functionality are all implemented using a simple Python script. In this tutorial, learn how to extend a Python script to add the ability to configure the area of system memory used by the TMC-ETR for storage of trace data.

The Arm CoreSight Trace Memory Controller (TMC) device plays a crucial role in trace capture and storage in many modern Arm-based designs. When implemented as an Embedded Trace Router (ETR), the TMC stores trace data in the system memory of the target platform. Because memory maps can change and the location and amount of memory available to the TMC might not be constant across debug sessions, we have added a tab to the DTSL Options dialog which contains a number of controls specifically targeted at memory configuration for the TMC. We have also scripted the functionality that takes user-entered values from these controls, and use them to configure the TMC.

This example shows how to build the ETR tab in the existing Altera Cyclone V SoC platform configuration from scratch.

Figure 1-1: Debug configuration menu showing the DTSL scripting button



2. Adding a tab to the DTSL Options dialog

Every DTSL script can provide a `getOptionsList()` method to return an array of objects representing the controls in the DTSL Options dialog. Because this method needs to be called before any DTSL objects have been instantiated (i.e. before the debug session has started), the method must be declared as static. For a simple target platform the `getOptionsList()` method can return an array of controls directly, but for a more complex platform this can lead to a method that is large, complicated and difficult to read. The Altera Cyclone V SoC platform needs a variety of controls spread across a number of tabs, so for this platform the `getOptionsList()` method simply calls a number of other methods, each of which returns the control objects for a single tab in the DTSL Options Dialog:

```
    @staticmethod
    def getOptionList():
        return [
            DTSLv1.tabSet("options", "Options", childOptions=[
                DtslScript.getOptionCrossTriggerTabPage(),
                DtslScript.getOptionTraceBufferTabPage(),

                DtslScript.getOptionCortexA9TabPage(),
                DtslScript.getOptionSTMTabPage(),
                DtslScript.getOptionETFTabPage()
            ])
        ]
```

We can add a new method for ETR control, which just returns an object for an empty tab:

```
    @staticmethod
    def getOptionList():
        return [
            DTSLv1.tabSet("options", "Options", childOptions=[
                DtslScript.getOptionCrossTriggerTabPage(),
                DtslScript.getOptionTraceBufferTabPage(),

                DtslScript.getOptionCortexA9TabPage(),
                DtslScript.getOptionSTMTabPage(),
                DtslScript.getOptionETFTabPage(),
                DtslScript.getOptionETRTabPage()
            ])
        ]

    @staticmethod
    def getOptionETRTabPage():
        return DTSLv1.tabPage("etrtab", "ETR", childOptions=[
        ])
```

Every DTSL control takes at least two parameters. The first is a string that identifies the control to DTSL, and is used when retrieving the user-assigned value of the control. The second parameter is the display name of the control, as it will appear in the DTSL Options dialog. Additional attributes are added to the control as `name=value` pairs; here we are using the `childOptions` attribute to pass an array of controls that will appear inside the tab. This array is currently empty and we now need to add these controls.

3. Adding controls

We need a master checkbox that controls configuration of the ETR memory buffer, and use the defaultValue attribute to ensure that it is unchecked by default. The checkbox will guard access to all of its child controls - unless the box is checked the child controls will appear greyed in the DTSL Options dialog and their values cannot be changed:

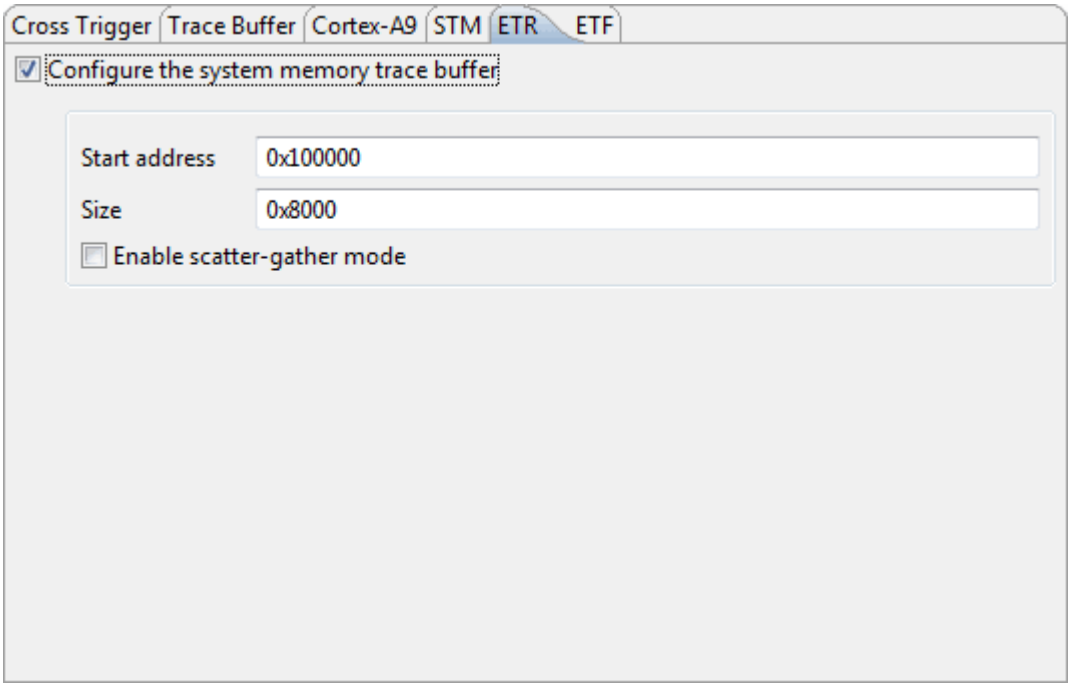
```
    @staticmethod
    def getOptionETRTabPage():
        return DTSLv1.tabPage("etrtab", "ETR", childOptions=[
            DTSLv1.booleanOption('etrBuffer',
                'Configure the system memory trace buffer',
                defaultValue = False,
                childOptions = [
                ]
            )
        ])
```

Now we can add the controls we need to configure the ETR memory buffer: edit boxes to control the start and the size of the buffer, and another checkbox to enable scatter-gather mode:

```
    @staticmethod
    def getOptionETRTabPage():
        return DTSLv1.tabPage("etrtab", "ETR", childOptions=[
            DTSLv1.booleanOption('etrBuffer',
                'Configure the system memory trace buffer',
                defaultValue = False,
                childOptions = [
                    DTSLv1.integerOption('start', 'Start address',
                        description='Start address of the system memory trace
buffer',
                        defaultValue=0x00100000,
                        display=IIntegerOption.DisplayFormat.HEX),
                    DTSLv1.integerOption('size', 'Size',
                        description='Size of the system memory trace buffer in
bytes',
                        defaultValue=0x8000,
                        display=IIntegerOption.DisplayFormat.HEX),
                    DTSLv1.booleanOption('scatterGather', 'Enable scatter-gather
mode',
                        defaultValue=False,
                        description='When enabling scatter-gather mode, the start
address of the on-chip trace buffer must point to a configured scatter-gather
table')
                ]
            )
        ])
```

As well as default values for each of the controls, we've used the description attribute to add a tooltip that will appear when the mouse is hovered over the control. We've also ensured that the edit boxes will deal only in hexadecimal numbers. The code that we've added is enough to create the new control tab and populate it with the controls that we need:

Figure 3-1: DTSL options dialog showing a custom ETR tab added using Python scripting



4. Using the controls values

So far we have added code to display additional controls in the DTSL Options dialog, but we haven't added any code to deal with the current values of the controls. The existing `optionValuesChanged()` method uses the `getOptionValue()` method to retrieve the current value of a control, using the DTSL identification string passed as the control's first parameter. We can add a section of code to the `optionValuesChanged()` method to retrieve the current values of the controls in our new tab, and pass them to existing methods provided by the object created to configure and manage the ETR device:

```
    def optionValuesChanged(self):
        # Set up the ETR buffer
        configureETRBuffer = self.getOptionValue("options.etrtab.etrBuffer")
        if configureETRBuffer:
            scatterGatherMode =
self.getOptionValue("options.etrtab.etrBuffer.scatterGather")
            bufferStart = self.getOptionValue("options.etrtab.etrBuffer.start")
            bufferSize = self.getOptionValue("options.etrtab.etrBuffer.size")
            self.ETR.setBaseAddress(bufferStart)
            self.ETR.setTraceBufferSize(bufferSize)
            self.ETR.setScatterGatherModeEnabled(scatterGatherMode)
```

5. Summary

That's it - that's all the code we need to add. This is a very simple example, adding a small number of controls to leverage some existing DTSL functionality, but the theory scales very well and can be used to add large numbers of tabs and controls with complex underlying functionality. The flexibility and extendibility can be used to give users the controls they need to customize a debug session and make the best of the capabilities of the target platform.