KINARM
insight in motion

Dexterit-E 3.9 Addendum

Creating Task Programs

CE

When installed in accordance with this manual and verified by a BKIN technician, the equipment described in this guide is in conformity with the relevant Essential Health and Safety Requirements of the following Directives:

- 2004/108/EC (relating to electromagnetic compatibility)
- 2006/95/EC (relating to electrical equipment designed for use with certain voltage limits)
- 2006/42/EC (on machinery)

140 Railway St, Kingston, Ontario, K7K 2L9, CANADA
Phone: +1.613.507.4393 Toll Free: +1.888.533.4393 Email: support@kinarm.com
Kinarm.com

Contents

1 Revision History

Revision	Description	Date	Approved By
1	Updated formatting style.	5-Jul-11	IEB
2	Added warning page.	12-Oct-11	IEB
3	New styles applied. Updated for Dexterit-E 3.2.	12-Dec-11	D. McLean
4	Updated for Dexterit-E 3.3	14-Nov-12	D. McLean
5	Updated for Dexterit-E 3.4	29-Jan-14	IEB
6	Updated for Dexterit-E 3.5	23-Jan-15	IEB
7	Updated for Dexterit-E 3.6	23-Mar-16	D. McLean
8	Updated for Dexterit-E 3.6.1	14-Sep-16	IEB
9	Updated for Dexterit-E 3.7	19-Jul-18	IEB
10	Updated for Dexterit-E 3.8	4-Nov-19	D. McLean
11	Updated for Dexterit-E 3.9. Conversion to DITA	22-Mar-21	IEB
12	Updated VC2017 installation instructions	19-Oct-21	IEB

2 Read This First

2.1 Conventions

The user information that accompanies the product uses typographical conventions to assist you in finding and understanding information.

- All procedures are numbered and all sub-procedures are lettered. You must complete the steps in the sequence in which they are presented to ensure success.
- Bulleted lists indicate general information and choices related to a function or procedure. They do not imply a sequential procedure.
- Control names and menu items or titles are spelled as they are on the system and they appear in highlighted text.
- Click or select means to move the pointer to an object or menu item and press the primary mouse button.
- Right-click means to point at an item and then press and immediately release the right mouse button without moving the mouse.

NOTE: Notes bring your attention to important information that will help you operate the product more effectively.

ATTENTION: These notes bring your attention to critical information that will help you operate the product more safely and effectively.

CAUTION: Cautions highlight ways you could injure yourself or the subject or damage your product and consequently void your warranty or service contract.

WARNING: Warnings highlight information vital to the safety of you, the operator, and the subject.

2.2 Audience

This information identifies the audience for the guide. These audience definitions are general in nature.

2.2.1 Researcher

This individual is responsible for creating, running, and analyzing custom task programs. This individual may also be responsible for interpreting the results of standard tests for research purposes.

2.3 Warnings and Cautions

2.3.1 Custom Task Creation

WARNING: Ensure the emergency stop button is readily accessible and/or keep a firm grip on the robot(s) when testing newly edited custom task programs. Care must be taken to reduce the risk of injury to the subject or operator, and/or damage to the Kinarm Lab. Mistakes in coding can result in unexpected system behaviour such as excessive forces and/or unstable loads.

2.4 Customer Comments

If you have questions about the user documentation, or to report an error in the documentation, contact us at:

- support@kinarm.com

Use the term "Documentation" in the subject line and include as much information as possible in the description included in your message.

2.5 Customer Service

If you encounter problems with your Kinarm Lab, contact **Kinarm Support** at:

- support@kinarm.com

Software downloads for your Kinarm Lab can be found at:

- https://www.kinarm.com/support/software-downloads/

3 In This Guide

This guide is intended for those users of Dexterit-E who wish to create customized Task Programs. This guide introduces:

- the concept of Task Programs;
- the basic code structure of Task Programs;
- numerous example Task Programs.

This guide assumes that the end-user is familiar with using Dexterit-E, including loading Task Programs, altering Task Protocols and the use of the Parameter Tables for the Task Protocols. It also assumes that the end-user is familiar with Simulink® and Stateflow®.

4 Custom Task Programs

One of the strengths of Dexterit-E is the flexibility provided to the end-user in customizing the software to meet their own needs. As outlined in the user's guide, Dexterit-E provides the flexibility of defining multiple Task Protocols for a given task. The purpose of this guide is to describe the next level of flexibility: creating a custom Task Program, which Dexterit-E can then use for running a novel task.

Task Programs are small programs used by Dexterit-E that define and control system behaviour during a single trial of a task. Task Programs are created to define system behaviour for a general class of tasks. For example, a Task Program could define that during a trial a target will turn on, once a subject reaches to that target it will turn off and another target will turn on, once the subject reaches to the second target it will turn off and then the trial is over (i.e. the general class of point-to-point reaching tasks). The details of the task (e.g. the target location, colour and number of trials) are not defined by the Task Program, but rather are specified as parameters in the Task Protocol and are defined through Dexterit-E's Microsoft Windows® based user interface.

Coding a Task Program involves MathWorks' Simulink and Stateflow toolboxes. These toolboxes provide a graphical programming environment in which Task Programs are developed and include many pre-built functions. We provide with Dexterit-E a custom library of Simulink blocks that are specific to Task Programs and greatly assist in rapid code development. For further information on Stateflow and/or Simulink, please read the MathWorks introduction to each of these toolboxes (http://www.mathworks.com/access/helpdesk/help/helpdesk.html).

5 Software Setup

This chapter provides a description of the software setup required to create and use custom Task Programs. Task Programs for Dexterit-E are created within the Simulink environment within MATLAB®. For an end-user to create a custom Task Program, MATLAB, various MATLAB toolboxes and a C/C++ compiler must all be installed on a computer running Windows. This computer does not need to be the computer running Dexterit-E. In addition, the end-user will need to install a set of Simulink libraries provided with Dexterit-E called the Dexterit-E Task Development Kit (TDK). While the TDK is included as part of Dexterit-E, MATLAB and its associated toolboxes are not included, nor is a C/C++ compiler.

5.1 Configuration of the Task Development Computer

5.1.1 Supported Dexterit-E, MATLAB and C Compiler Versions

We verify and support products for use with only specific combinations of versions of the Dexterit-E Task Development Kit (TDK), MATLAB and the C compilers supported by MATLAB. Other combinations of versions are NOT supported and attempts to use other combinations will result in errors when attempting to compile tasks with the TDK. The table below shows which versions of MATLAB and C compilers have been verified to work with Dexterit-E.

Table 5-1: Dexterit-E, MATLAB and C Compiler Version Compatibility

Supported MATLAB Release	Supported C Compilers	Supported Dexterit-E TDK Versions
R2015a SP1 (64 bit)	Microsoft Windows SDK 7.1	3.6 or greater
R2019b update 7 or later (64 bit)	Microsoft Visual Studio 2017 Community Edition	3.9 or greater
http://www.mathworks.com **NOTE:** By default MathWorks will offer you only the latest version of MATLAB; be sure to select a supported version	Download from website. Requires a login user name and password.	Download from website. Requires a login user name and password.

NOTE: Dexterit-E 3.9 only supports Robot Computer PN 11103 or higher.

5.1.2 Installing the C Compiler

The first step in setting up a computer for building Custom Task Programs is to install the required C Compiler. MATLAB R2015a SP1 requires the Windows SDK 7.1 to be installed as the C compiler and MATLAB R2019b requires Microsoft Visual Studio 2017 Community Edition to be installed as the C compiler. The instructions for installing the C compiler depends on the version of Windows that it is being installed on.

Install the C Compiler: Windows 7, R2015a SP1

In order to use the Windows SDK 7.1 as the C Compiler, MATLAB R2015a SP1 requires .NET 4.0.

NOTE: The installer for Windows SDK 7.1 requires that .NET 4.0 be installed PRIOR to installing the Windows SDK 7.1.

NOTE: Later versions of .NET (e.g. .NET 4.5 or .NET 4.6) are NOT supported by MATLAB R2015a for the purposes of building tasks.

1. To determine if .NET 4.0 is installed, go to Control Panel and look at installed programs.

   NOTE: .NET Framework 4.0 may be listed as ".NET Framework 4", and is listed separately from .NET Framework 4.5, etc.

2. If .NET 4.0 needs to be installed, you will have to remove any later versions first.
3. You may re-install later versions of .NET if necessary after installing .NET 4.0.
4. Download the .NET 4.0 installer from the Kinarm Support website.
5. Download the Windows SDK 7.1 installer from the Kinarm Support website.
6. Ensure that any previously-installed versions of Microsoft Visual C++ 2010 redistributable are 10.0.30319 or OLDER (i.e. lower number).
7. Go to Control Panel and look at installed programs.

   Check BOTH the x86 and x64 versions. If the version number is higher than 10.0.30319, then remove those redistributables before proceeding

8. Proceed with the SDK installation by clicking on setup.exe and select the defaults.
9. If you have errors installing Windows SDK 7.1, please see *Section: Troubleshooting Windows SDK 7.1 Installation*. If you want to have Visual Studio Express SP1 installed as well as Windows SDK 7.1 then be sure to read this page first: http://www.mathworks.com/matlabcentral/answers/96611. There is a patch to the 7.1 SDK that you will need if you want both installed at the same time.

Install the C Compiler: Windows 10, R2015a SP1

1. Download the Windows SDK 7.1 installer from the support page of the Kinarm Support website.
2. Uninstall any Visual C++ 2010 redistributables (x86 and x64) from the Windows 10 Control Panel.
3. From the installer, run \Setup\SDKSetup.exe directly (i.e. do not run \setup.exe).

Install the C Compiler: Windows 10, R2019b

1. Download the most recent version of Microsoft Visual Studio 2017 Community Edition from the Microsoft web site
2. Run the installer
   a. Select **Desktop development with C++**
   b. Ensure that at least the following options are selected:
   – Visual C++ core desktop featuresVC++ 2017 toolset (x86, x64)Windows 10 SDK

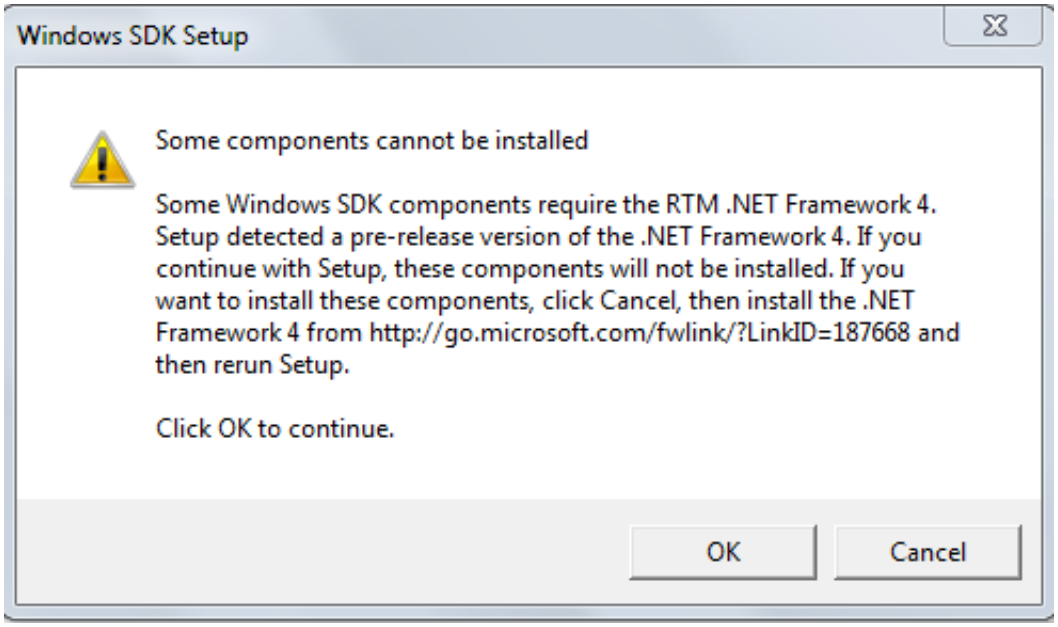
Troubleshooting Windows SDK 7.1 Installation

Various issues may occur during installation of the Windows SDK. The following are some common errors and possible resolutions.

Setup detected a pre-release version of the .NET Framework 4.

If you see this error message it indicates that you have an incompatible version of the Microsoft .NET libraries on your computer. You will need to remove all of the newer .NET library versions in order to allow the installation to continue.

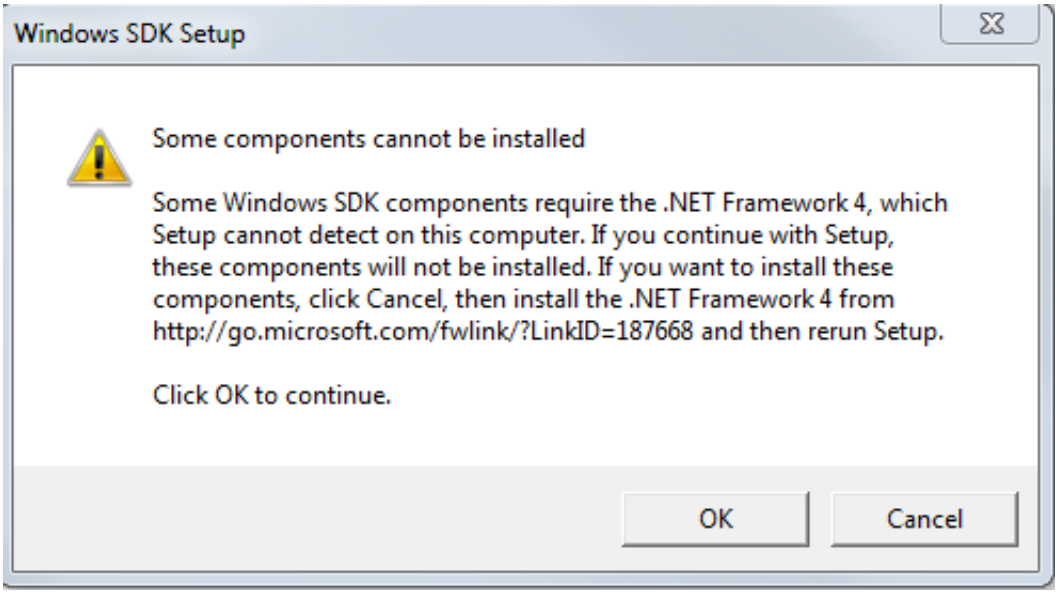
Figure 5-1: SDK Installation Error: Pre-release of the .NET Framework Detected



1. From the Start menu open the Control Panel.
2. In the Control Panel select **Programs and Features** -> **Uninstall a program**.
3. Scroll through the list of installed programs and find all versions of Microsoft .NET Framework.
4. Uninstall each version of the .NET framework you find.
5. Install the Microsoft .NET framework 4.0 and continue with the installation of the Windows SDK 7.1 as per *Section: 5.1.2 Installing the C Compiler*.

Microsoft .NET Framework is Not Detected

Figure 5-2: SDK Installation Error: .NET 4 Framework Cannot Be Detected



The above error means that the Microsoft .NET Framework 4.0 is not installed.

1. Install the Microsoft .NET framework 4.0 and continue with the installation of the Windows SDK 7.1 as per *Section: 5.1.2 Installing the C Compiler*.

Installation Failed – Please refer to Samples\Setup\HTML\ConfigDetails.html

Figure 5-3: SDK Installation Error: Installation failed.

This error can be difficult to see because it is buried in the text of a dialog that looks like

the installation is complete. This error is most often caused by a program that needs to be removed.

1. From **Start** open the *Control Panel*.
2. In the *Control Panel* select **Programs and Features** -> **Uninstall a program**.
3. Scroll through the list of installed programs and find all versions of Microsoft Visual C++ 2010 Redistributable (there should be at least a x64 and an x86 version).
4. Uninstall each of the Microsoft Visual C++ 2010 redistributables you find.
5. Continue with the installation of the Windows SDK 7.1 as per *Section: 5.1.2 Installing the C Compiler*

A Newer Version of Windows SDK is Installed

You may have a more recent version of Windows SDK installed. This could interfere with the installation of Windows SDK 7.1. Please remove the newer SDK version(s) and try installing again.

Other Errors

If Windows SDK 7.1 was installed while .NET 4 was not installed (i.e. if various warnings that the SDK installer provides were missed), then errors will occur during subsequent MATLAB builds. If this situation occurs, then the SDK will have to be completely uninstalled and then reinstalled after first installing .NET 4.0.

NOTE: If this situation has happened, using the installer to repair or modify the installation of the SDK will not work. You must completely uninstall the SDK, install .Net 4.0, then reinstall the SDK.

5.1.3 MATLAB, Simulink and Simulink Real-Time Installation

In order to create custom Task Programs for use with Dexterit-E, the end-user must install MATLAB R2015a SP1 or R2019b along with the following MATLAB toolboxes:

Table 5-2: Required MATLAB Toolboxes

R2015a SP1 (64-bit) or R2019b
• Simulink® • Stateflow • MATLAB Coder™ • Simulink Coder™ • Simulink Real-Time™

MATLAB and the supporting toolboxes must be purchased directly from MathWorks and downloaded from the MathWorks website.

ATTENTION: By default the MathWorks' website will attempt to have you install the most recent version of MATLAB. You must manually select one of the supported versions listed above prior to download.

ATTENTION: For MATLAB R2019b, Update 7 or higher must be used.

Kinarm Patch for MATLAB R2015a SP1

In order to use MATLAB R2015a SP1 an additional patch is required. This patch is installed automatically as part of installing the TDK. If MATLAB is re-installed AFTER the TDK is installed, then the TDK will need to be re-installed in order to re-apply the patch.

5.1.4 Set MATLAB Up to Use the Correct C Compiler

The end-user must manually configure MATLAB to use the correct C compiler. This configuration can be done as follows:

1. In MATLAB type `slrtsetCC -setup`
2. Select the C compiler.
   - For MATLAB R2015aSP1 select Windows SDK 7.1
   - For MATLAB R2019b select Microsoft Visual Studio 2017

   Occasionally other software can cause problems with MATLAB finding the correct compilers. If you have tried the instructions above and your task fails to compile then at the MATLAB prompt you can try the following commands:

   a. `mex -setup` and choose Windows SDK 7.1

   b. `slrtsetCC('VisualC')`

3. In MATLAB type `mex -setup`
4. Select the compiler.
   - For MATLAB R2015aSP1 select the Windows SDK 7.1
   - For MATLAB R2019b select Microsoft Visual Studio 2017

5.1.5 Install the Task Program Development Kit

A set of Simulink libraries that are known as the Task Development Kit (TDK) are provided for your Kinarm Lab. These need to be installed in order to create custom Task Programs with MATLAB.

1. Download the most recent version of the TDK from the **Kinarm Support** website.
2. Run the installer.

   If MATLAB R2015a SP1 is detected, then the installer will install a patch that is required for MATLAB R2015a SP1. This process can take up to a minute while MATLAB is opened and updated.

   NOTE: If MATLAB R2015a SP1 is installed or re-installed after installing the TDK, then the TDK will need to be re-installed to ensure that this patch is applied.

3. Select which versions of MATLAB you would like to use and click **Next**.

   As part of running the TDK installer, all installations of supported MATLAB versions will be found. You will be prompted to select which versions of MATLAB you would like to use with this version of the TDK (i.e. so that the MATLAB path is updated correctly).

   This process can take up to a minute while each version of MATLAB is opened and updated.

Dexterit-E and Task Development Kit Version Compatibility

Due to the underlying structure of the software, the versions of Dexterit-E and the Task Development Kit (TDK) used to build Task Programs must be matched. In particular, the minor version must match, and the patch version of Dexterit-E must be equal to or greater than the patch version of the TDK, as per the following table. Refer to subsequent sections on issues for updating a Task Program from one version of the TDK to another.

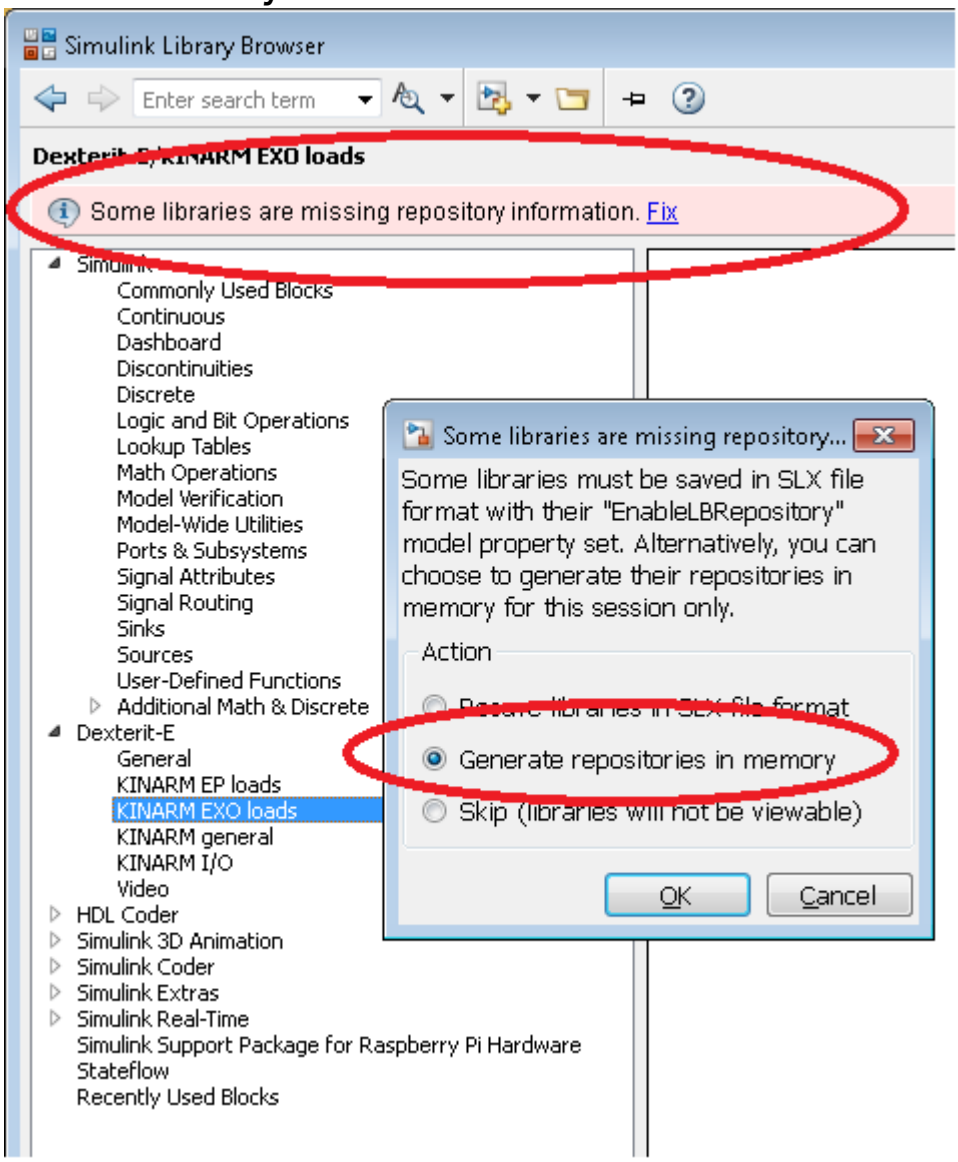
Table 5-3: Dexterit-E and TDK Version Compatibility

Dexterit-E Version	Compatible TDK Versions
3.x.n	3.x.m (where m <= n)

Fix TDK Libraries

It is common to receive a warning in the library browser indicating that the TDK libraries are in the wrong format. The warning will appear as a pink banner in the Simulink Library Browser when you select one of the Dexterit-E libraries.

Figure 5-4: MATLAB Library Error



This problem can be corrected by following these instructions:

1. In the pink error banner, click **Fix**.
2. Select **Generate repositories in memory**.
3. Click **OK**.

5.1.6 Verify the MATLAB Environment

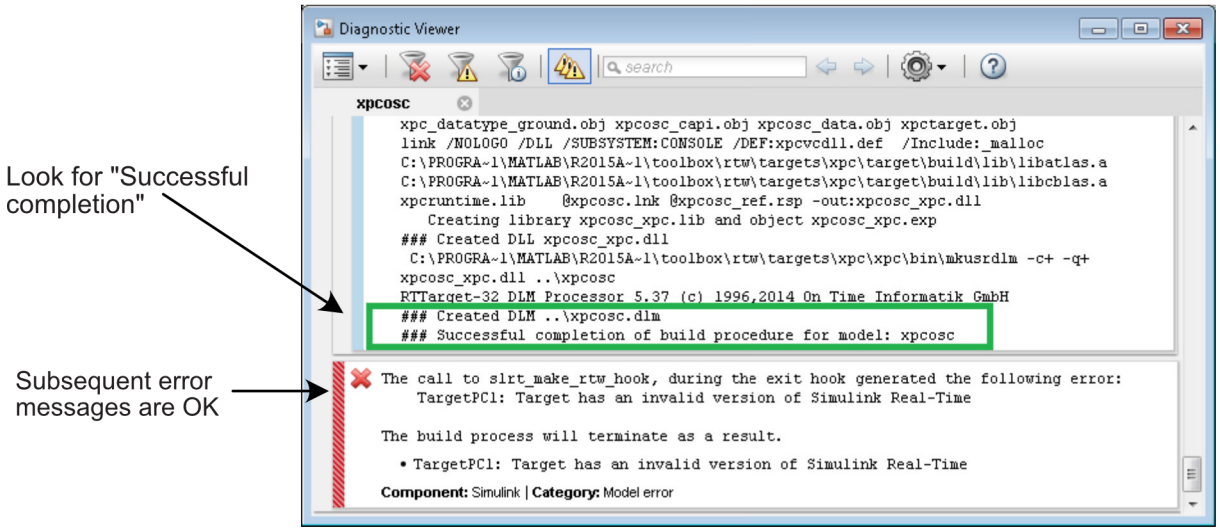
The following steps need to be followed to verify that the MATLAB build environment has been set up correctly.

1. Close and restart MATLAB.
2. At the MATLAB prompt, type `dexcheck -all`.

   The output should contain the following:
   - All Dexterit-E required libraries found for MATLAB R201xx
   - SLRT Compiler found
   - mex compiler found
   - SUCCESS checking for components required to build custom tasks
3. Correct any errors before proceeding.
4. At the MATLAB prompt, type `xpcosc`.
5. When the model opens, compile it using the **Build** button (or press **Ctrl**+**B**).
6. Confirm that the build succeeds.

   A “Successful completion...” message should be generated, as shown in the figure below.

Figure 5-5: Successful Build of xpcosc



7. If setting up the MATLAB environment in preparation for training by Kinarm, take a screen shot of this window pressing **Alt+Prt Sc** and send the image to Kinarm Support.

5.2 Update the Robot Computer OS Version

The Robot Computer of a Kinarm Lab uses an Operating System (OS) provided by MATLAB called - Simulink Real-Time OS (SLRT). The Robot Computer needs to boot to the version of SLRT that is associated with the version of MATLAB you are using to build custom Task Programs. Kinarm Labs are typically shipped with the most recent SLRT version supported by Dexterit-E. For Kinarm Labs or Robot Computers purchased in 2017 or later (PN 13107 or greater), Dexterit-E has the ability to change the version of SLRT installed on the Robot Computer. For further details see the Dexterit-E User Guide's section on the Robot Computer OS version.

For earlier systems, the software mentioned above will not work, and a bootable CD/DVD will need to be created. Supported boot CD/DVD images for SLRT can be found on the **Kinarm Support** website.

ATTENTION: Dexterit-E does not support the use of other boot images.

For more information on creating SLRT boot disks, please refer to the SLRT documentation from the Kinarm Support website.

To create an SLRT boot disk, complete the following procedure:

1. Log on to the **Kinarm Support** website's download section.
2. Under **Software Downloads** find the **MATLAB & Compilers** section. Choose the correct version of the ISO image to download and download it.

   An ISO image is an archive file of a CD/DVD, with the file extension ".ISO".

   If you are using MATLAB R2015aSP1 then you will need one of the SLRT 6.2 ISOs
3. Once you have downloaded the correct ISO image, unzip it, and burn it to a CD or DVD as an ISO image.

   Burning an ISO image is not the same procedure as copying and burning other files to a CD/DVD, or backing up a CD/DVD, nor is it the same as "drag 'n drop" to a CD/DVD. You will need 3rd party software (e.g. provided by your CD/DVD burner's manufacturer to burn an ISO image). Please refer to your software utility's help file for details on burning an ISO image.

   The burned CD or DVD can then be used as a boot disk for the Robot Computer.

6 From Concept to Code: Converting an Idea into a Task Program

This chapter provides an overview of creating a Task Program from the descriptive outline of a task. It begins with a description of what a Task Program is, followed by the settings and configuration parameters necessary within Simulink, and then a walk-through example of going from a descriptive outline of a task to actual code that can be used by Dexterit-E.

In order to properly understand this chapter and the remainder of this guide, the end-user must be familiar with using Dexterit-E. In particular the end-user must know how to load, edit and create Task Protocols for an existing Task Program. Please refer to the Dexterit-E User Guide for more information.

6.1 Task Programs

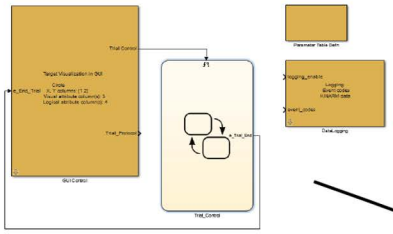
Task Programs are programs used by Dexterit-E that define and control the system behaviour that can occur during a single trial of a task. For example, a Task Program could define:

- During a trial a target will turn on.
- Once a subject reaches to that target it will turn off and a second target will turn on.
- Once the subject reaches to the second target it will turn off.
- The trial is over.

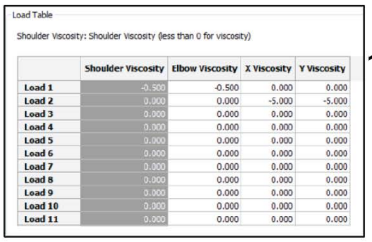
In this example, note that the Task Program does not define certain parameters, such as the location, size and colour of the two targets. Nor does it define how many trials will occur or how these parameters change from trial to trial. These parameters are part of what is called a Task Protocol. Together, the Task Protocol and Task Program define everything that will occur during a task. Thus a more explicit definition of a Task Program is that it defines system behaviour for a general class of tasks, this example being a general class of point-to-point reaching task. The separation of Task Program from the parameters stored in a Task Protocol means that multiple Task Protocols can be associated with a single Task Program. The manner in which Task Programs relate to Task Protocols is shown schematically in *Figure: 6-1* below.

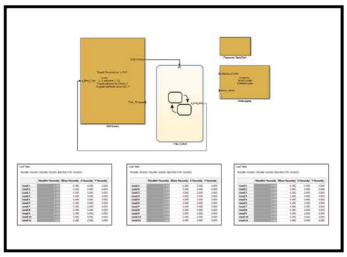
Figure 6-1: How do Task Protocols and Task Programs Relate?

Task Program
- implements a generic paradigm
- parameter values are unspecified



Task Protocol
- specifies the parameter values that are required by the Task Program





Both are required to run a task!

6.2 The Programming Environment: Simulink and Stateflow

Programming a new Task Program involves MathWorks' Simulink and Stateflow toolboxes. Simulink is the graphical programming environment in which Task Programs are developed and provides a graphical representation of data flow in the task. Stateflow is a graphical design tool that is part of the Simulink environment. Stateflow is used to develop event-driven state machines called Stateflow Charts that control the flow of behaviour during a task. Simulink and Stateflow were chosen to provide a stable programming environment with as much flexibility for the end-user as possible.

Simulink comes with many of its own pre-made functions, or blocks. We provide a Kinarm specific library of Simulink blocks (referred to as the Task Development Kit, or "TDK") that are specific to Dexterit-E's Task Programs and assist in rapid code development. See *Section: 5.1.5 Install the Task Program Development Kit* for details on installing the TDK.

If you are unfamiliar with Stateflow and/or Simulink, we recommend going to the MathWorks' website (`http://www.mathworks.com/help/index.html`) or select **Help** from the MATLAB menu, and learn more about Simulink and Stateflow. Understanding Simulink and Stateflow is essential to understand the code for the Task Programs shown below and this user guide has been written with the assumption that the end-user has a basic knowledge of Stateflow and Simulink. For questions related to which version of MATLAB to use, and therefore which version of Simulink/Stateflow, please refer to *Section: 5.1.1 Supported Dexterit-E, MATLAB and C Compiler Versions*. In particular, the sections listed below are of particular significance.

6.2.1 Suggested Simulink Help Sections to Read

- Getting Started
- Simulink Basics
- Creating a Model
- Using the Embedded MATLAB Function Block
- Chapter summary for all other chapters (to be aware of other possible chapters of interest)
- Blocks - By Category/Alphabetical List (for reference)
- Examples (for reference)

6.2.2 Suggested Stateflow Help Sections to Read

- Getting Started
- Stateflow Concepts
- Stateflow Notation
- Stateflow Semantics
- Creating Stateflow Charts
- Extending Stateflow Chart Diagrams
- Defining Events and Data
- Using Actions in Stateflow
- Semantic Rules Summary
- Control Chart Execution Using Temporal Logic
- Chapter summary for all other chapters (to be aware of other possible chapters of interest)
- Examples (for reference)

The above suggestions are a minimum suggested set of reading. Which other sections should be read will be end-user and need dependent.

6.3 Task File Types: SLX, DLM, MLDATX, DTP

There are three main file types associated with a task.

- SLX - <your_task>.slx files are Simulink model files that contain the source code for Task Programs. These files are used by MATLAB and Simulink when creating and

building a custom Task Program. These files are not used by Dexterit-E directly. This file is created by the user within the Simulink environment.

NOTE: MathWorks is transitioning Simulink model file types from MDL to SLX. While both R2015a and R2019b support both file types, future versions of MATLAB will support only SLX.

- DLM - <your_task>.dlm files are the compiled or built version of a Task Program for R2015a. These files are used by Dexterit-E when running a task. This file is created by Simulink as the result of a build.
- MLDATX - <your_task>.mldatx files are the compiled or built version of a Task Program for R2019b. These files are used by Dexterit-E when running a task. This file is created by Simulink as the result of a build from MATLAB R2019b.
- DTP - <your_task_protocol>.dtp files contain all the details for a Task Protocol. These files are used by Dexterit-E when running a task. This file must reside in the same directory as <your_task>.dlm or <your_task>.mldatx. DTP files are created within Dexterit-E.

6.4 Initial Configuration of <your_task>.slx

In order for a Simulink model file to be compatible as a Task Program for Dexterit-E, various configuration parameters need to be set in the Simulink model file. For end-users who are creating their own Task Program, we recommend starting with an existing `<Task Program>.slx` and modifying it to suit their own needs. Examples of Task Program code can be downloaded from **Kinarm Support**. This approach will ensure that various settings in `<your_task>.slx` are set correctly. You can then remove those blocks from the Task Program that are not needed, and add in other blocks as needed.

Alternatively a new Simulink model file can be used, with the parameters set as described below. To set the required parameters for a new Simulink model file, from within Simulink select **Simulation** -> **Configuration Parameters** and set the parameters to match those in the dialogs shown in *Figure: 6-2* through to *Figure: 6-5*.

The required step time is 0.00025s (i.e. 4 kHz).

Figure 6-2: Configuration Parameters: Solver

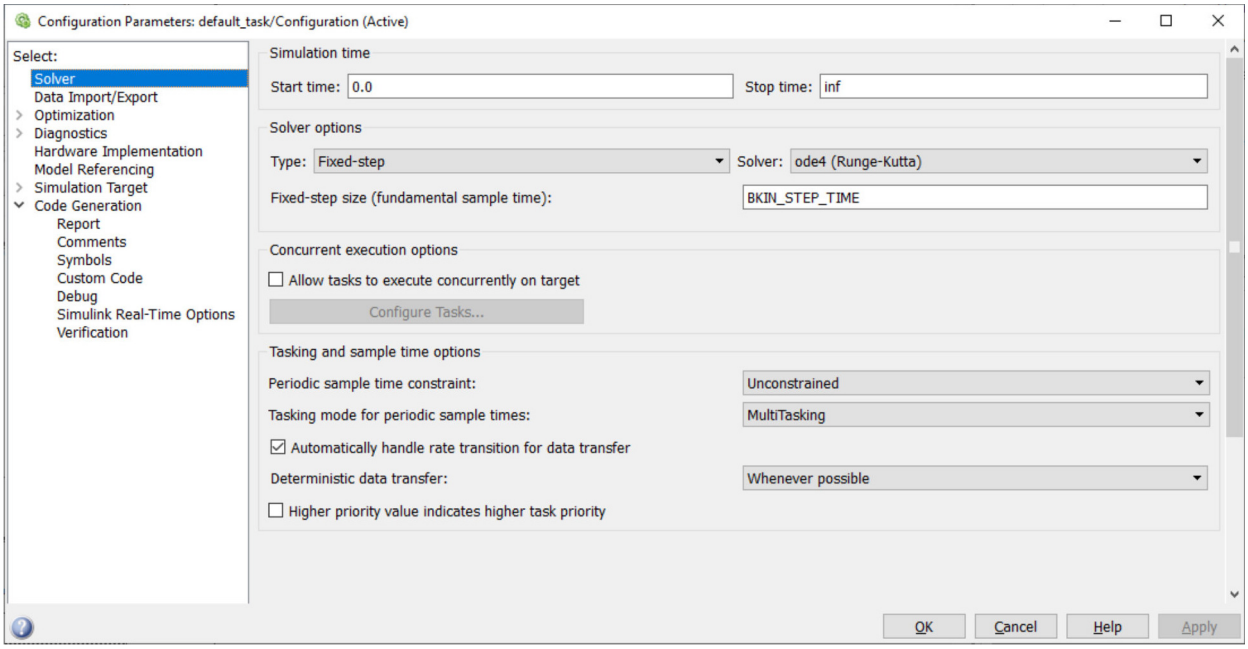


Figure 6-3: Configuration Parameters: Hardware Implementation

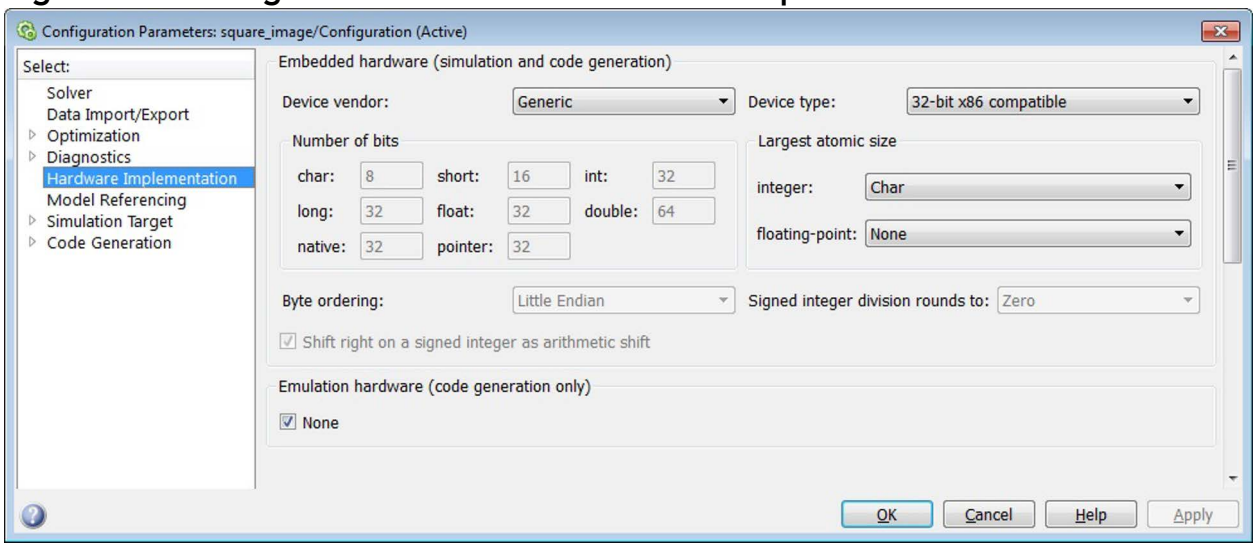


Figure 6-4: Configuration Parameters: Code Generation

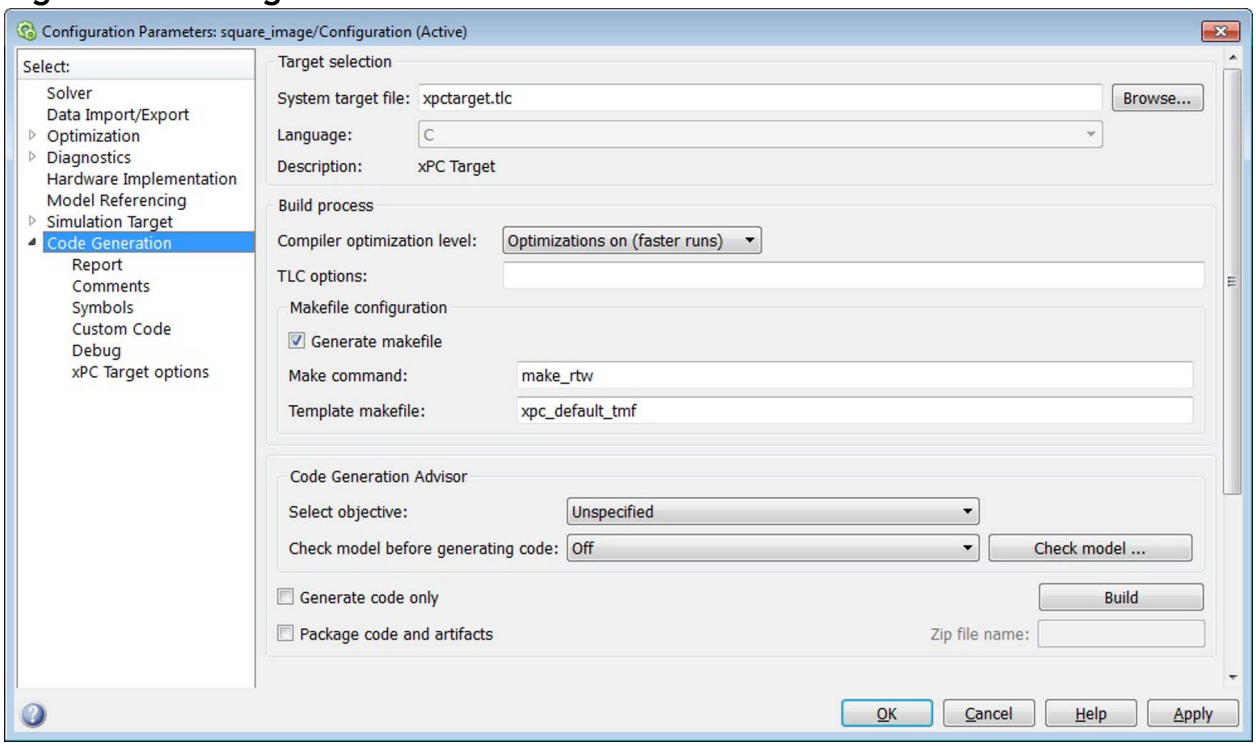
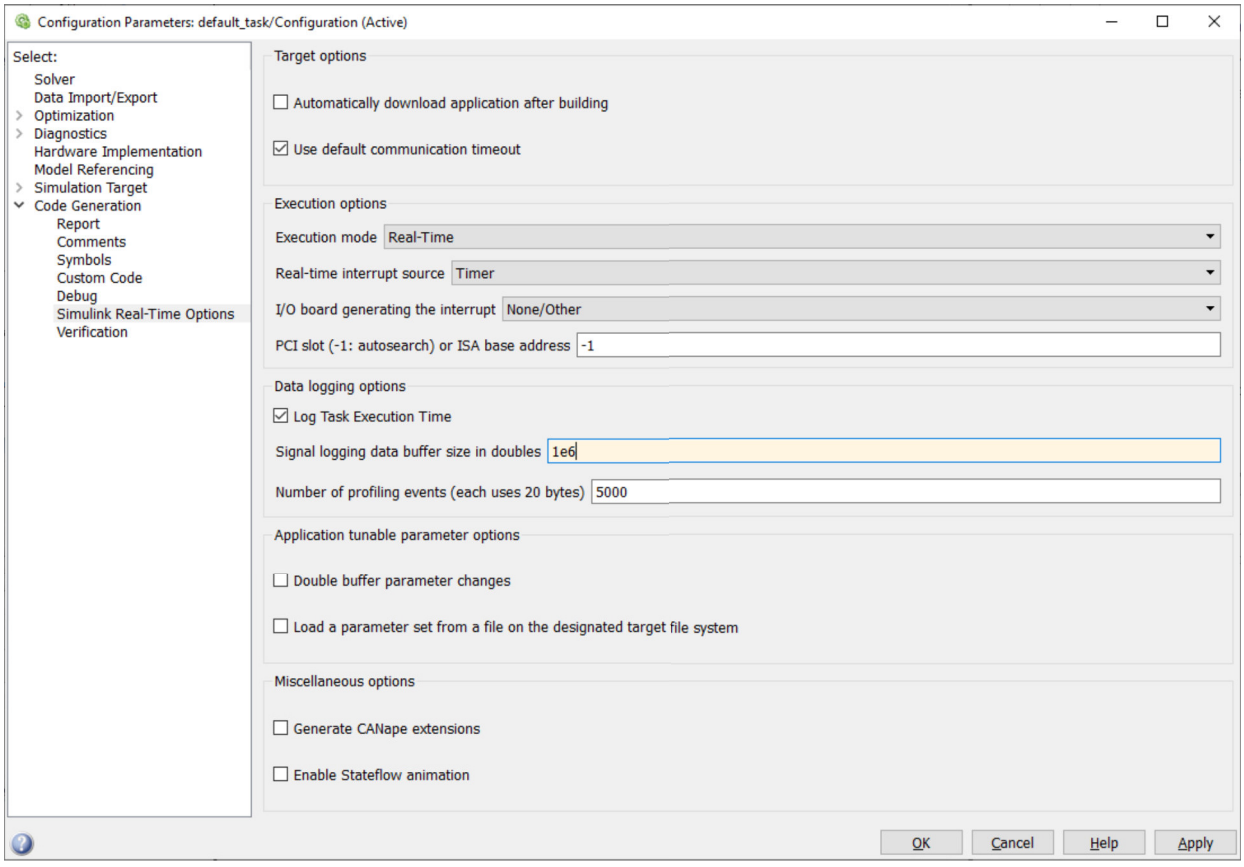


Figure 6-5: Configuration Parameters: SLRT Options

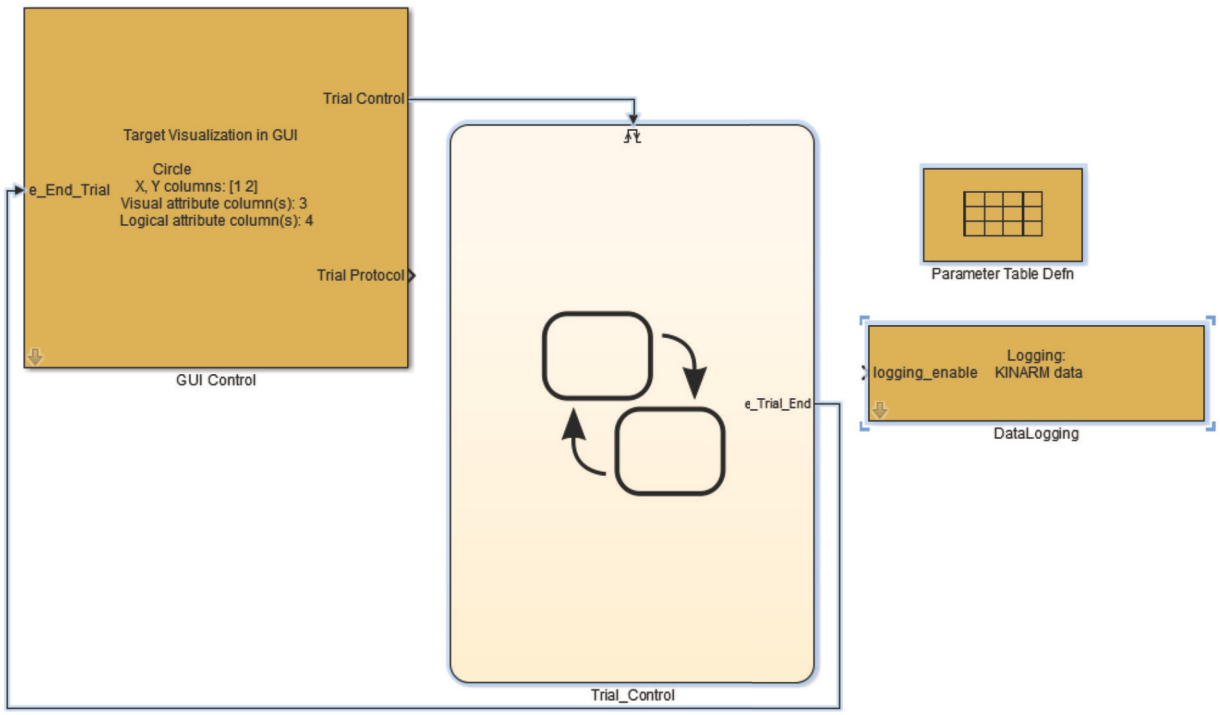


6.5 Simulink Blocks Required for <your_task>.slx

Once you have added the Simulink parameters described in *Section: 6.4 Initial Configuration of <your_task>.slx*, four additional Simulink blocks, required by Dexterit-E, need to be added. These blocks must all be present in the top level of the Simulink model.

- `GUI Control`
- `Trial_Control`
- `Datalogging`
- `Parameter Table Defn`

Figure 6-6: Required Simulink Blocks for a Task Program



The `GUI_Control` block also controls the timing and sequence of trials. It outputs the appropriate Trial Protocol for each trial. It receives feedback from the Stateflow chart in the form of an `e_End_Trial` Stateflow event; it can, optionally, receive a `Repeat_Trial` input (not used in this example).

In the middle of the Stateflow chart called `Trial_Control`.This controls the the behaviour of the system during a individual trial. Various inputs provide Stateflow events and data that can be used to drive the Stateflow chart. Various outputs from the Stateflow chart can be used to control robot behaviour, visual stimuli, and drive external Stateflow events. The Stateflow chart can also interact with other parts of the Simulink model.

The `Data_Logging` block logs all data to be saved by the Task Program. This includes Kinarm Lab-related data, such as position and velocity, as well as Task Event Codes and analog input data (not shown). Data logging only occurs when the `logging_enable` input is set to 1 (true) and the trial is running (so that data logging pauses when the trial pauses).

The `Parameter Table Defn` block allows Dexterit-E to read a Task Program so that Dexterit-E knows how to create the Task Protocol parameter tables for that Task Program. Double-clicking the Parameter Table Defn block brings up a GUI in which you can define the columns to be used in the following Task Program tables:

- Target Table
- Load Table
- LP Table

The `Parameter Table Defn` also includes the definitions for the Task Wide Parameters, Task Event Codes, and Task Control Buttons.

Follow the instructions in *Section: 6.5.1 Add Required Simulink Blocks to <your_task>.slx*.

6.5.1 Add Required Simulink Blocks to <your_task>.slx

After you have set your new Simulink model up as per *Section: 6.4 Initial Configuration of <your_task>.slx*, you will need to add the required blocks to your model, and then set up the Stateflow chart to interact with these blocks.

1. From the Simulink Library Browser open the `Dexterit-E -> General` library
2. Drag and drop the required blocks into your model:
   - `GUI Control`
   - `DataLogging`
   - `Parameter Table Defn`
3. In the Simulink Library Browser find the Stateflow library.
4. Drag and Drop a Stateflow Chart block into your model.
5. Rename the Stateflow Chart to `Trial_Control`.
6. Open the Stateflow Chart by double-clicking on it in Simulink.
7. Select **Chart** -> **Add Inputs and Outputs** -> **Event Input from Simulink**.
8. In the new *Event* dialog, set the **Name**, **Scope**, **Port** and **Trigger**.
   - **Name**: e_clk
   - **Scope**: Input
   - **Port**: 1
   - **Trigger**: Rising
9. Select **Chart** -> **Add Inputs and Outputs** -> **Event Input from Simulink**.
10. In the new *Event* dialog, set the **Name**, **Scope**, **Port** and **Trigger**.
    - **Name**: e_ExitTrialNow
    - **Scope**: Input
    - **Port**: 2
    - **Trigger**: Either
11. Select **Chart** -> **Add Inputs and Outputs** -> **Event Input from Simulink**.
12. In the new *Event* dialog, set the **Name**, **Scope**, **Port** and **Trigger**
    - **Name**: e_Trial_End
    - **Scope**: Output
    - **Port**: 1
    - **Trigger**: Either
13. Add a signal wire from the GUI Control block's Trial Control output to the top of the Trial Control block
14. Add a signal wire from the Trial Control block's e_Trial_End output to the GUI Control block's End_Trial input.

There are almost always numerous other Simulink blocks present in the final Task

Program. Which ones will differ for each Task Program. *Section: 6.9 Which Simulink Blocks to Add?* provides more information, as does *Section: 10 Examples of Task Program Code*. For more information on the functionality of individual Simulink blocks available in the Dexterit-E library, click **Help** for each block from within Simulink.

6.6 Where to Start

Creating a Task Program starts with a descriptive outline of a task. There needs to be a description of what kinds of things will happen during the task, when they will happen, if they are conditional and if so, conditional upon what. For example, details that need to be specified include anything that will affect or interact with the subject (e.g. a target will be presented or a load applied), or alternatively something that might be required for subsequent data analysis by the operator (e.g. a record of a Task Event Code when the subject reaches a target). Typically the description needs to include all the variations of what can happen (e.g. sometimes there will be one target shown, sometimes two targets shown) and when and how these variations will occur (e.g. one target for first 10 trials, two targets for next 10 trials). Often these descriptions are purely text-based, although they can include figures or flow charts.

As an example to be used throughout this chapter, we will next describe a sample 'centre-out' reaching task. The rest of this chapter will use this example as a basis for going from concept to code.

6.7 Centre-out Reaching Task

6.7.1 Task Description

In this sample task, a subject will be presented with a centre target followed by a random selection of one of 8 peripheral targets. The subject will be instructed to reach to the first target and hold at that target. After holding at the target for a specified period of time, the target will turn off and a peripheral target will appear; the subject is to reach quickly and accurately to the second target. Once the peripheral target is reached, there is a delay before the target turns off and another delay until the start of the next trial. There will be 10 blocks of 8 trials, each block consisting one reach to each of the peripheral targets, presented in a random order.

6.8 Task Breakdown: Task Program or Task Protocol

Before the transformation from description to code can take place, we need to clarify what part of the task description goes into the Task Program, and what goes into the Task Protocol. Because changes in the Task Program require rebuilding and redeploying the Task Program, it is advantageous to minimize the number of times that process will have to take place. Choosing those parameters that are most likely to be changed from one variation of a task to another to be defined in the Task Protocol versus the Task

Program will help achieve this goal. As described previously, a Task Program defines and controls the system behaviour that can occur during a single trial of a task, so clarifying the definition of what constitutes a trial is the first step in the task breakdown. This step is then followed by choosing which parameters go into the Task Protocol and which are hard-coded into the Task Program.

6.8.1 Trial Definition

A typical task is composed of a defined number of trials. Sometimes each trial is identical, but typically there are predefined changes in task behaviour from trial to trial. The key part of this step is choosing what portion of the task is being repeated over and over again. Often the choice is obvious, but that is not necessarily the case. In the centre-out reaching task described above, one possible choice for what constitutes a trial includes the following sequence of events:

1) Presentation of a centre-target.
2) Subject reaches to the centre target.
3) Presentation of a peripheral target (after the required hold period at first target).
4) Subject reaches to the peripheral target.
5) Pause before turning off second target.

6.8.2 Choice of Task Protocol Parameters

Task Protocol parameters are set in Dexterit-E using the Task Protocol Parameter Tables. Their values are not defined in the Task Program. However, because they are used by the Task Program, we need to create variables for them for the Task Program. The choice of what to make a Task Protocol parameter needs to be made at this juncture. As mentioned above, having a parameter be set in the Task Protocol versus having it hard-coded in the Task Program provides a significant advantage: it allows us to quickly change task behaviour without having to rebuild the Task Program.

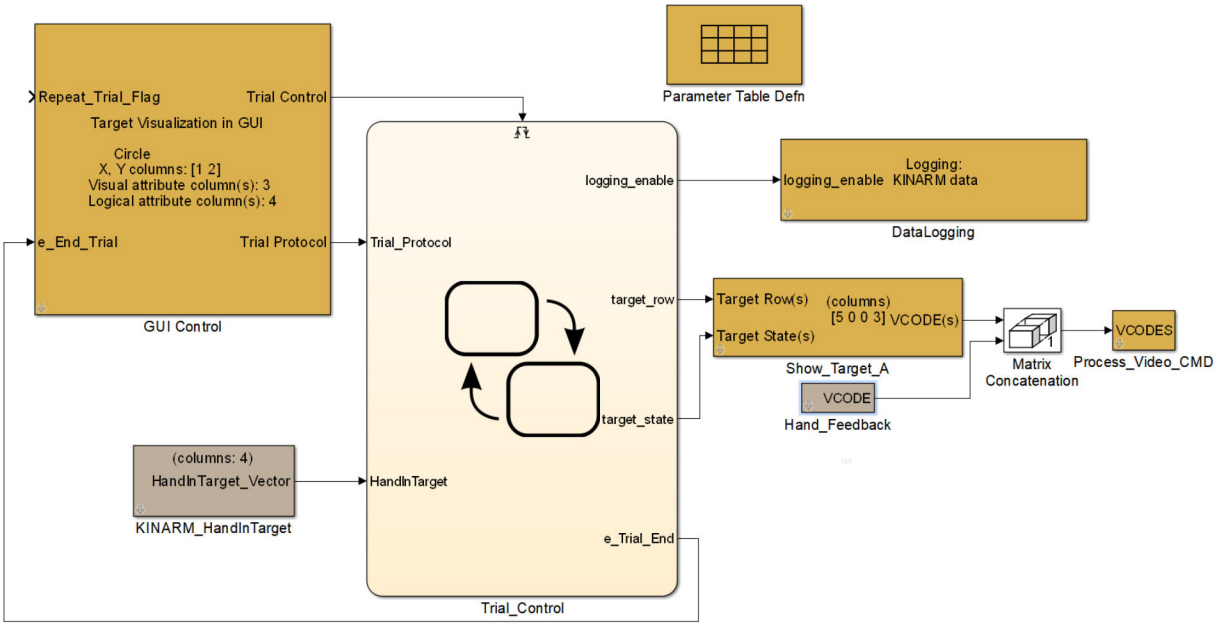
A Task Program only defines what will happen during a single trial. Any parameter that changes from trial to trial should be part of the Task Protocol. In addition, it is useful to consider which parameters might be different in a future experiment. Any parameter that has a reasonable likelihood of changing its value can, and probably should, be set as a Task Protocol parameter.

For this example, the size, colour and location of both the centre and peripheral targets will be set as Task Protocol parameters. In addition, the length of time required to hold at the centre target will be a parameter. A more complicated task will often be many more Task Protocol parameters. For example, if a Task Program included forces or loads then parameters defining the forces or loads would typically be set as Task Protocol parameters. In our Centre-out Reaching example, it would be possible to hard-code the size, colour and location of the centre target, because it is constant for all trials. However, by choosing it to be a Task Protocol parameter, we ensure that future variations of this task can include different target sizes, colours and locations, without having to rebuild the Task Program.

6.9 Which Simulink Blocks to Add?

We can now add other Simulink blocks as necessary for this task. As shown below, in a screen-shot of the final Simulink code for this task, there are numerous coloured blocks, all of which are included as part of the Task Development Kit: gold-coloured blocks, which are blocks specific to Dexterit-E, and brown blocks, which are blocks specific to Kinarm Labs. Each of these blocks is described briefly after the figure. For more information on any of these blocks, please refer to the examples in *Section: 10 Examples of Task Program Code*, or right-click on any Simulink block and select **Help**.

Figure 6-7: Simulink Code for Centre-out Reaching Task Example



The `GUI Control`, `Trial_Control`, `DataLogging` and `Parameter Table Defn` blocks were described in *Section: 6.5 Simulink Blocks Required for <your_task>.slx*. The `Trial_Control` block now has additional inputs and outputs. The inputs are used to drive the Stateflow Chart within the `Trial_Control` block, and the outputs are used to drive the other Simulink blocks. How to create these inputs and outputs is shown in *Section: 6.10 The Model Explorer - Adding Input and Outputs to a Stateflow Chart*.

The `KINARM_HandInTarget` block provides feedback about which targets the hand is in. This block allows the Stateflow chart to act on that information as will be shown below. For systems with a Kinarm Gaze-Tracker there is a `KINARM_GazeInTarget` block that will report if the subject's gaze is touching a target.

Visualization of targets and hand feedback is controlled by the remaining three coloured blocks: `Show_Target`, `Hand_Feedback` and `Process_Video_CMD`. The `Show_Target` block creates a `VCODE` output which is a visual command to be processed by the `Process_Video_CMD` block to draw a target. The `Hand_Feedback` block creates a `VCODE` for the hand feedback to be shown to the subject. The `Process_Video_CMD` processes the video commands and outputs them in an appropriate format to the computer controlling the video.

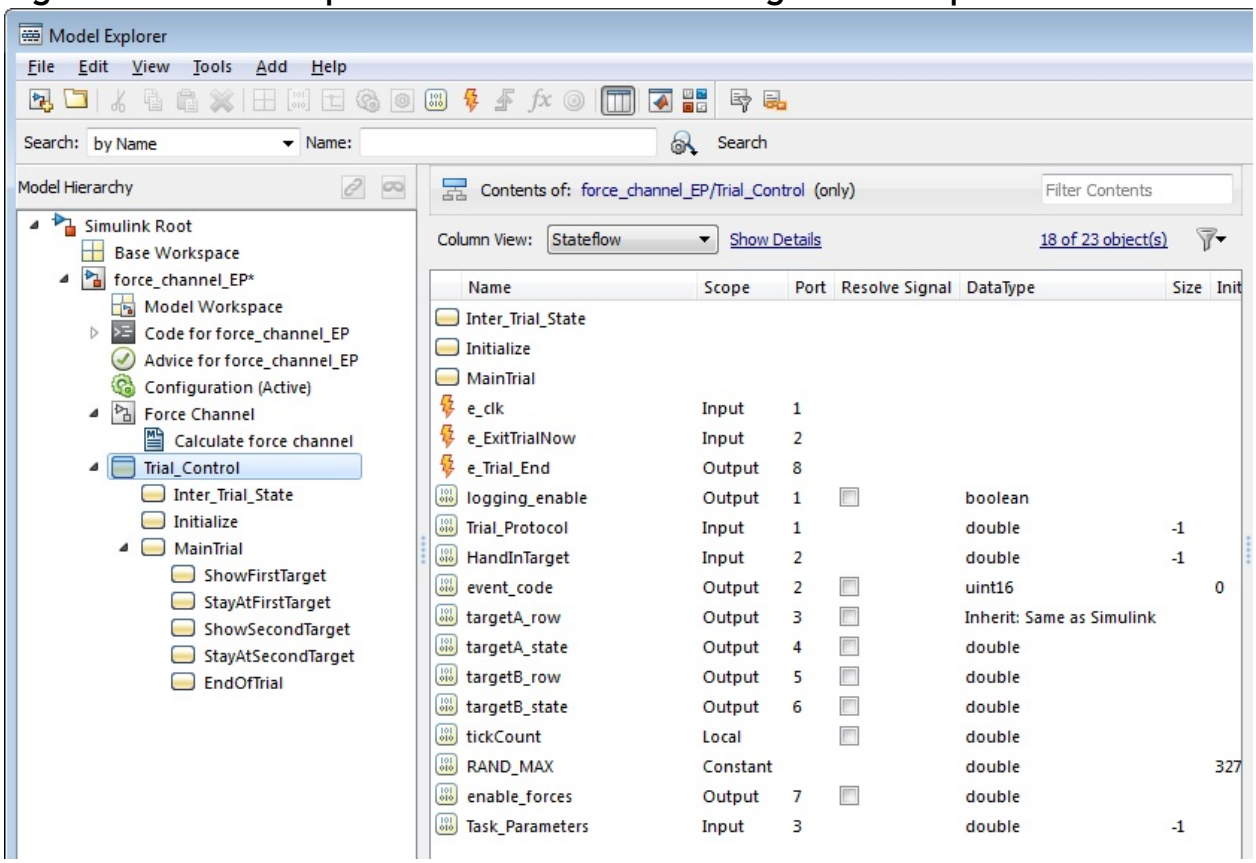
For more detailed information on each of these blocks, open up the relevant block within Simulink and click **Help**.

NOTE: The `GUI Control` block, `DataLogging` block, `Parameter Table Defn` block, and `Trial Control` Stateflow chart must all be in the "top-level" of the Task Program.

6.10 The Model Explorer - Adding Input and Outputs to a Stateflow Chart

There are two ways to add inputs and outputs to a Stateflow Chart. As shown in the figure below and described in *Section: 6.5 Simulink Blocks Required for <your_task>.slx*, one can use the Stateflow Editor (i.e. the **Add** menu option). Alternatively, inputs and outputs can be defined using the Model Explorer. The advantage of understanding the Model Explorer is that it presents all previously defined inputs and outputs, allowing them to be modified. From within Simulink or Stateflow, select **View** -> **Model Explorer** to open it up. A screen-shot of the Model Explorer for this example task is shown below.

Figure 6-8: Model Explorer for Centre-out Reaching Task Example



In the left-hand pane is the model hierarchy, in which the Stateflow Chart `Trial_Control` has been selected. In the right-hand pane are the inputs, outputs, constants and local variables that have been defined for the `Trial_Control` Stateflow chart. There are two types of inputs, outputs and local variables: data (matrix symbol) and Stateflow events (lightning symbol). Data contain numerical values for reference in the Stateflow diagram (i.e. floating point or integer values, as defined), whereas events drive the Stateflow diagram execution (see *Section: 6.2 The Programming Environment: Simulink and Stateflow* for suggested reading if the concept of a Stateflow event is not clear). For more information on how to use the Model Explorer, please refer to the Simulink Help documentation.

Whenever an input or output is defined for a Stateflow Chart in the Model Explorer, it has an effect on the Simulink block containing the Stateflow Chart. Data inputs appear as input ports on the left side of the Stateflow Chart block (e.g. see `HandInTarget` input of `Trial_Control`, in *Figure: 6-7*). Data outputs appear as output ports on the right side of the Stateflow Chart block (e.g. see `logging_enable` output of `Trial_Control`, in *Figure: 6-7*). Input Stateflow events must be wired to a single input port at the top of the Stateflow Chart block (e.g. see signal from `Trial Control` output of `GUI Control` block in *Figure: 6-7*); all input Stateflow events share the same input port. Output Stateflow events appear on the right side of the Stateflow Chart as output ports, and are listed after the Data output ports (e.g. see `e_Trial_End` output of `GUI Control` block in *Figure: 6-7*).

The input and output Stateflow events listed in the figure above must be defined for all Stateflow Charts in a Task Program, and must be wired as shown in *Section: 6.5 Simulink Blocks Required for <your_task>.slx*. Creating these required input and output Stateflow events using the Stateflow Editor has already been described in *Section: 6.5 Simulink Blocks Required for <your_task>.slx*. The meaning and use of the Stateflow events is described here.

- **e_clk** – This input Stateflow event must be on port 1, and have a defined trigger of Rising. This event occurs every 1.0 ms, as long as the Task Program is running (i.e. as long as the task has started and is not paused). It originates in the `GUI Control` block.
- **e_ExitTrialNow** – This input Stateflow event must be the on port 2, and have a defined trigger of Either. This event occurs whenever an operator clicks **Pause** and Dexterit-E has been set to pause the trial immediately. This event can be used by the end-user to define what will happen in the Task Program when this event occurs. It originates in the `GUI Control` block.
- **e_Trial_End** – This output Stateflow event can be on any output event port, but must have a defined trigger of Either. This event is used by the end-user to signal when a trial is over, so that the Task Program can update itself for the next trial. It terminates in the `GUI Control` block.

NOTE: All inputs, outputs, constants and variables to be used in a Stateflow chart, including both data and Stateflow events, must be defined in the Model Explorer. If other input Stateflow events are muxed with the GUI Control block events (i.e. `e_clk` and `e_ExitTrialNow`), the GUI Control block events must remain the first two signals of the muxed signal.

6.11 "Centre-out Reaching Task": From Text Description to Stateflow Chart

In this section, the text description of what will happen during a single trial of this task will be converted to Stateflow code. This process will be done in several steps to help clarify the different ideas of what needs to happen for this transformation.

6.11.1 Text Description of Task

Based on the description of the task and our subsequent definition of a trial for this task, we have parsed out a generic description of a trial for this task. It is important to recognize in the following description that no specific reference to shape, size or colour of targets has been made, nor any reference to number of trials.

During a single trial of this task, a target will be presented to which the subject must move. Once the subject reaches the target, a specified time period will pass after which the target will disappear and a peripheral target will appear. Once the subject reaches to the peripheral target, a specified time period later the target will disappear and the trial is over.

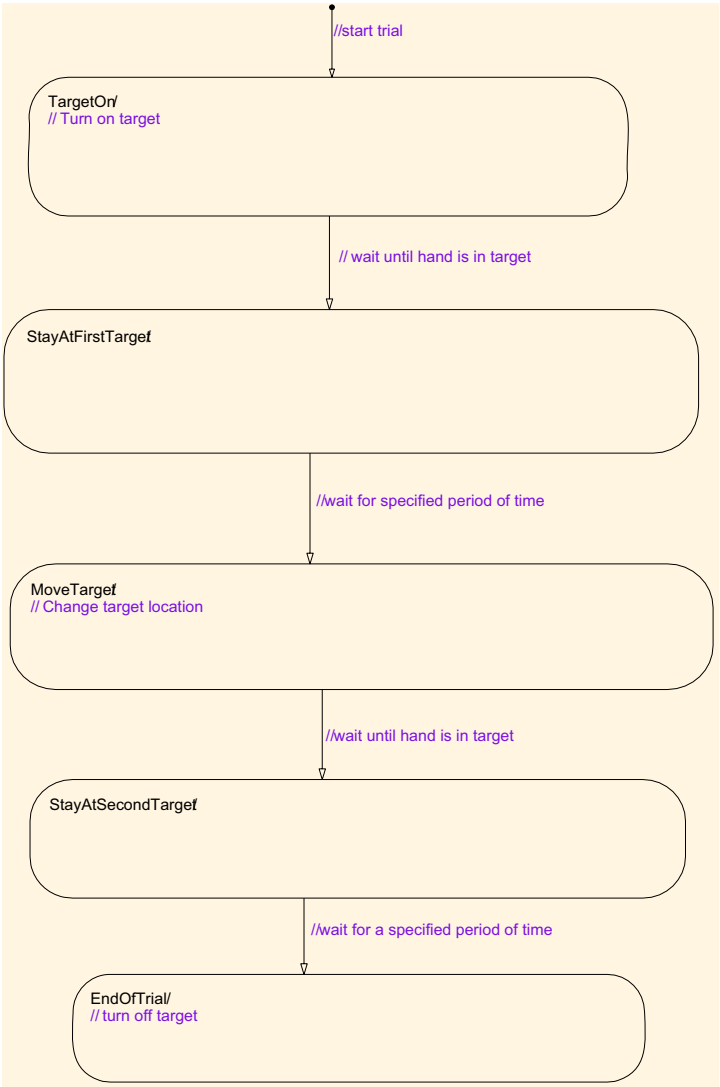
NOTE: No specific reference to shape, size or colour of targets has been made, nor any reference to number of trials, etc.

6.11.2 Conversion to a Simplified Flow Chart

The figure below shows a flow chart version of what was just described in text of what will happen during a single trial for this task. In this flow chart, the diagrammatic conventions of Stateflow have been used. States are represented by ovals, transitions between states are represented with arrows, and small circles known as 'connective junctions' provide a way to connect multiple transitions together between states. In Stateflow, the system can generally only exist in one state at a time, and the system must be in one of the defined states (i.e. the system cannot enter and stay at a connective junction). Changes from one state to another can only occur if there is a transition between the states and the conditions required for that transition are true.

In this step of the conversion process, we are not yet going to introduce any code. Instead, each state includes a name and a description of what will happen when that state is entered. Transitions include a description of the conditions that must be true in order for the transition to happen.

Figure 6-9: Simplified Flow Chart for Centre-out Reaching Task Example



There are five states defined in this example Stateflow Chart, each of which is briefly described below.

- **TargetOn** – When this state is entered, the first target will be turned on. The system will stay in this state until the condition "wait until hand is in target" is true, whereupon the system will switch to the `StayAtFirstTarget` state.
- **StayAtFirstTarget** – Nothing happens when this state is entered. Once a specified period of time has elapsed, the system will switch to the `MoveTarget` state.
- **MoveTarget** – When this state is entered, the target will be moved to the peripheral position (which is functionally identical to turning off the first target and simultaneously turning on the second target). The system will stay in this state until the

condition "wait until hand is in target" is true, whereupon the system will switch to the `StayAtSecondTarget` state.

- **StayAtSecondTarget** – Nothing happens when this state is entered. Once a specified period of time has elapsed, the system will switch to the `EndOfTrial` state.
- **EndOfTrial** – When this state is entered, the target is turned off.

6.11.3 Other Task Program Controls to Add to Simplified Flow Stateflow Chart

Although the flow chart above defines the gross behaviour of what can happen during a typical trial, there are other issues that typically need to be addressed to ensure proper Task Program control. Some of these include:

- an initializing state;
- when to record data;
- time limits on transitions to ensure that an infinitely long trial never occurs;
- a Stateflow event to indicate the end of trial (required by Dexterit-E);
- inter-trial timing.

Each of these issues is addressed in the figure below. As before, no code is introduced yet, only the descriptions.

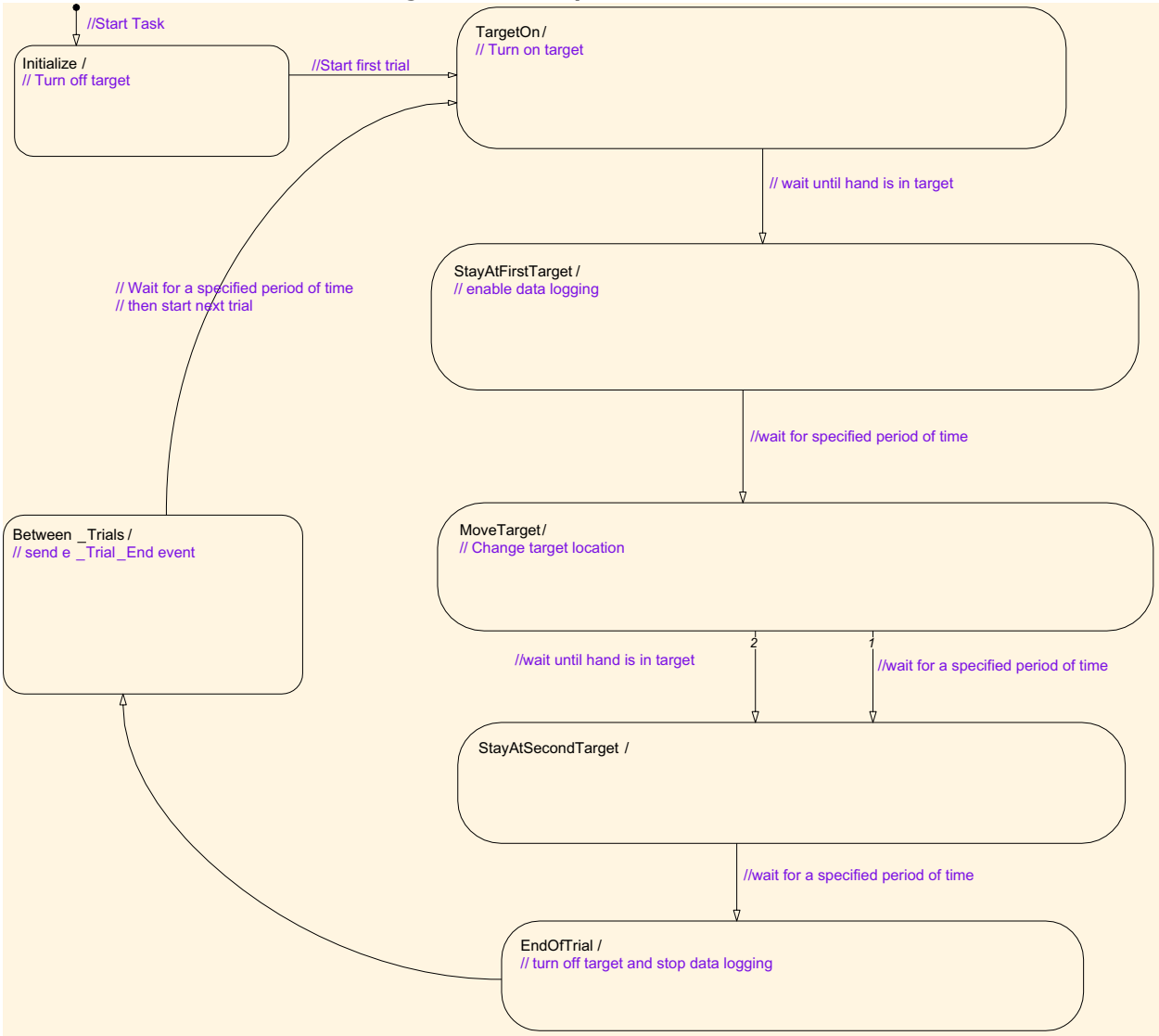
When a flowchart is first entered, there needs to be a default starting location. It is good practice to initialize parameters at this point to ensure that the system is starting in the correct configuration.

The choice of when to record data will depend upon the purpose of the task. In this example, we have chosen to start recording data when the subject reaches the first (i.e. centre) target. An advantage of this choice versus starting to record when the target is first presented is that if a subject decides to pause before 'starting' the trial, e.g. to talk to the operator, then the amount of irrelevant data saved is reduced. A disadvantage would be for the situation in which the initial movement to the centre target was of interest. Data recording is stopped in this example when the target lights turn off.

A time limit was added to each of the transitions that did not previously have a time limit to ensure that an infinitely long trial did not occur. This addition ensures stable behaviour during a task for unexpected conditions (e.g. if there was a single target location that the subject could not reach to, but it was important to still collect the other data without having to manually pause the task and force it to the next trial).

Inter-trial timing has been added, as well as an inter-trial state. The inter-trial state is necessary for Dexterit-E to function properly: it provides a time when the values of the Task Protocol parameters for one trial can be changed to those of the next trial.

Figure 6-10: Adding Task Program Controls to a Simplified Flow Chart for Centre-out Reaching Task Example



There are now seven states in the Stateflow Chart. The new additions are described below:

- **Initialize** – When the Task Program is first loaded, it needs to enter a default state, which in this case is the `Inititialize` state. The only action to be taken when this state is entered is to ensure that the target is initially off. There are no conditions for the transition leading to the `TargetOn` state, so the system immediately switches to the `TargetOn` state.
- **TargetOn** – This state is the same as before.
- **StayAtFirstTarget** – Data logging is now initiated when this state is entered (i.e. only data from this point on will be recorded for subsequent data analysis).
- **MoveTarget** – There are now two possible conditions for exiting this state (if either is true, then the transition will occur). There is now a timing condition: once a speci-

fied period of time has elapsed, the system will switch to the `StayAtSecondTarget` state if the "wait until hand is in target" condition has not yet become true.

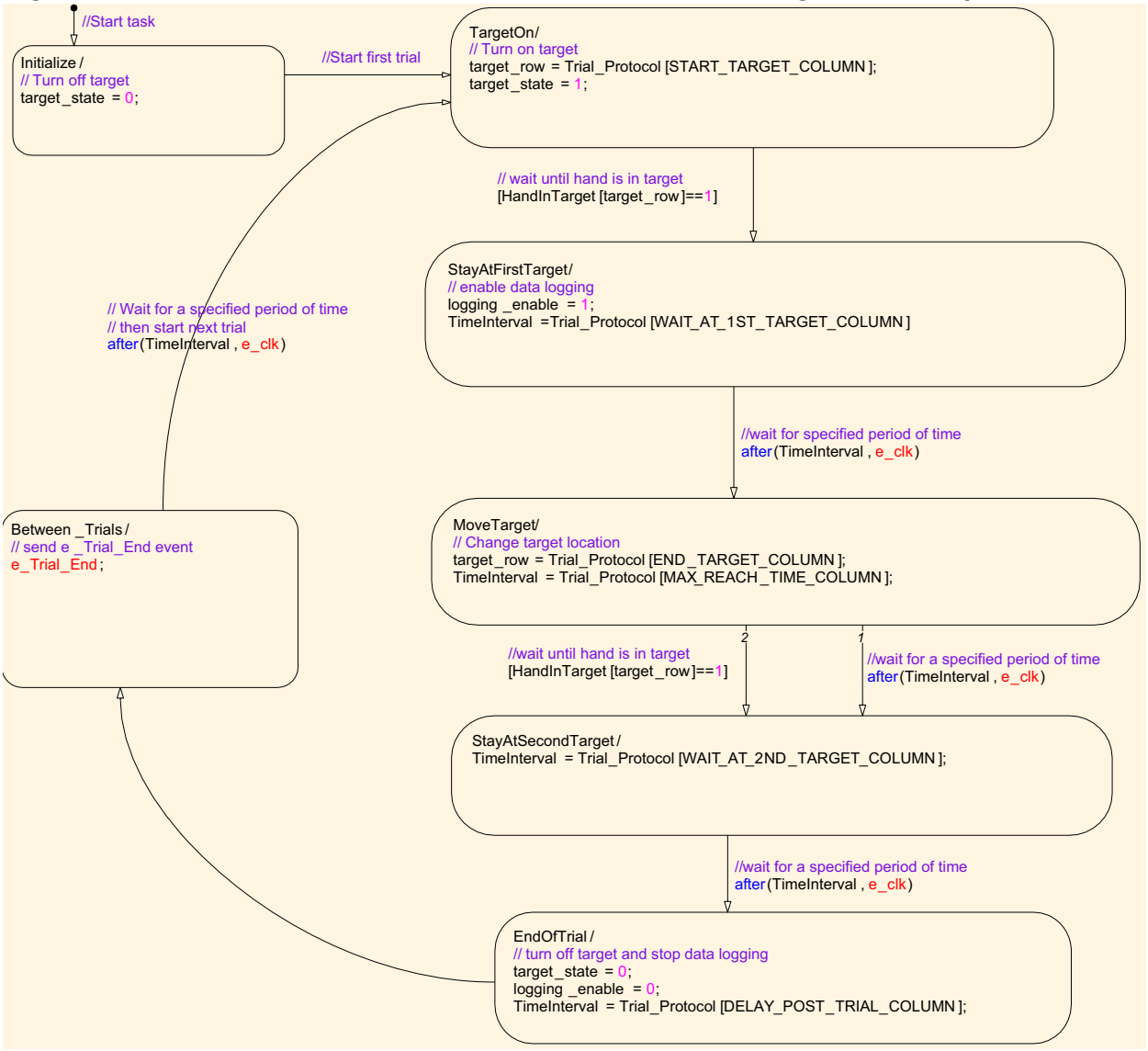
- **StayAtSecondTarget** – Nothing new occurs in this state.
- **EndOfTrial** – This state now transitions to a new `Between_Trials` state. There is no condition on this transition, so it occurs immediately.
- **Between_Trials** – This state is a place holder that occurs between trials. Its primary functions are (a) to send a Stateflow event out of the Stateflow chart to signal to the Task Program that the trial is over and (b) to provide a single time step delay, which allows the Task Program to update the `Trial_Protocol`for the next trial. Exiting of this state back to the `TargetOn` state for the next trial occurs after a specified time delay.

As a Task Program gets more complex, there are numerous other Task Program controls that an end-user may want to add at this pre-code stage: for example, Task Event Codes for post-data collection analysis (see *Section: 10.1 Task Event Codes*) or **Pause** button control (see *Section: 6.13 Including Pause Button Functionality*).

6.11.4 Final Stateflow Chart

In order to get to a usable Stateflow Chart, the necessary Stateflow code needs to be added to the states, transitions and descriptions that we have thus far provided. In the following figure, we have added the relevant code to implement the described actions and Stateflow events. The code acts upon inputs to the Stateflow chart, and affects outputs from the Stateflow chart. In the code, these inputs and outputs appear as variables. For example, `Trial_Protocol` is an input and `target_row` is an output. There are also various local variables used, for example `TimeInterval` and constants, such as `START_TARGET_COLUMN` (defined in the `Parameter Table Defn` block).

Figure 6-11: Final Stateflow Chart for Centre-out Reaching Task Example



There are 7 states in this Stateflow Chart, each of which will be described briefly.

- **Initialize** – When the Task Program is first loaded, it needs to enter a default state, which in this case is the `Inititialize` state, which simply ensures that the `target_state=0` (i.e. target is off)
- **TargetOn** – The actions of this state result in a target being drawn by Simulink as soon as this state is entered. The choice of target is defined by the `target_row` and`target_state` outputs of this Stateflow chart (these outputs drive the `Show_Target` block in the Simulink diagram shown previously in *Figure: 6-7*). `Trial_Protocol` is a vector input to the Stateflow chart and `START_TARGET_COLUMN` is a constant pointing to the appropriate index in the `Trial_Protocol`where the `target_row` for this target is stored. The `TargetOn` state is maintained until the required condition of "hand is inside the target" comes true, as defined by the [`HandInTarget[target_row]==1`] condition.

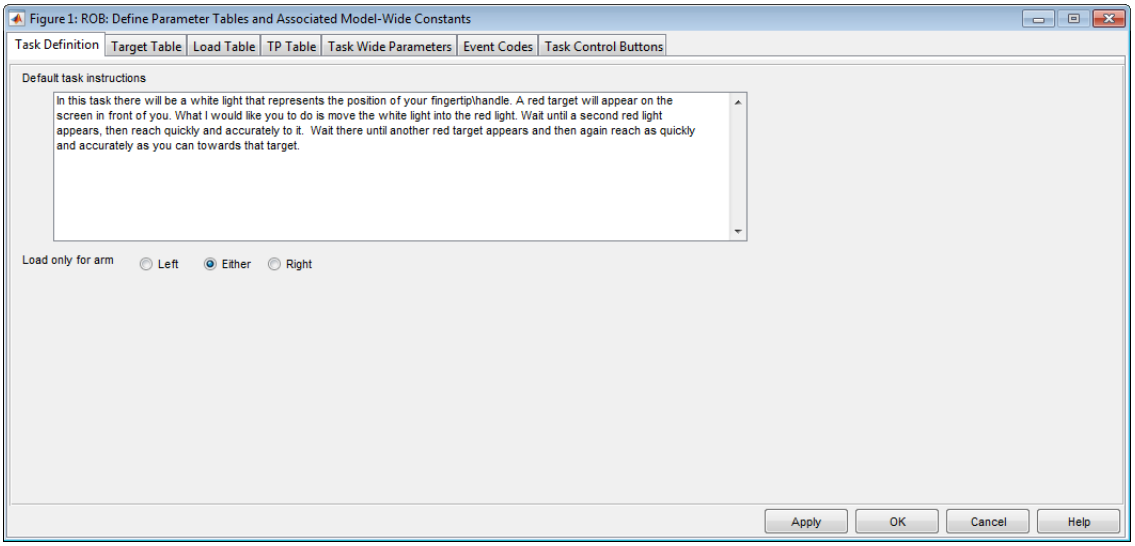
`HandInTarget` is an input to the Stateflow chart which comes from a Simulink block that checks to see whether the hand is in any of the targets. If the hand is in the target associated with the present `target_row`, then the condition will be true and the Stateflow chart proceeds to the next state.

- **StayAtFirstTarget** – This state is entered when the hand has reached the target. One action occurs upon entering this state: `logging_enable=1` is set. This setting tells Dexterit-E to start recording data (`logging_enable` is an output of the Stateflow chart, see *Figure: 6-7*). The system will then stay in this state until the exiting transition is true, which in this case is a time delay defined by the `after` function. The `after` function in Stateflow counts and waits for a prescribed number of Stateflow events. In this example, the prescribed number of Stateflow events to count is `TimeInterval`, and the events being counted are `e_clk` events. For Task Programs, `e_clk` events are input Stateflow events that occur every 1 ms. (The `e_clk` Stateflow event is sent by the `GUI_Control`block, and passed to the Stateflow chart through the event input located at the top of the Stateflow chart - see *Section: 14.10 GUI Control Block Input Events and Output Events* for more information).
- **MoveTarget** – The actions of this state turn off the first target and turn on the second target. This action occurs simply by changing the value of `target_row` within the Stateflow chart. Because `target_row` is an output connected to the `Show_Target` block in Simulink, this change produces an effect in Simulink (see *Figure: 6-7*). There are now two ways for the task to proceed: the task will wait until the hand is in this new target or until a specified period of time has elapsed before proceeding to the next state. The latter is defined by the `after(TimeInterval, e_clk)` condition.
- **StayAt2ndTarget** – As with the `StayAtFirstTarget`state, this state simply adds a time delay before proceeding to the next state.
- **EndOfTrial** – At the end of the trial, the `TimeInterval` variable is updated for the desired between-trial delay (used when exiting the subsequent state). Because there is no condition attached to the output transition from this state, the transition occurs automatically when the next Stateflow event occurs (`e_clk` events occur every 1 ms, so this transition will happen on the next `e_clk` event).
- **Between_Trials** – This state is a place holder that occurs between trials. It has two primary functions. First it sends a Stateflow event (`e_TrialEnd`) out of the Stateflow chart to signal to the Simulink model that the trial is over. Second it provides a single time step delay, which allows the Simulink model to update the `Trial_Protocol` for the next trial. Transitioning from this state to the `TargetOn` state occurs after a time delay, determined by the `TimeInterval` updated in the `EndOfTrial` state.

6.12 Using the Parameter Table Defn Block

When an end-user double-clicks the `Parameter Table Defn` block they will be presented with an interface that allows them to define all of the editable parameters that they wish to have available for a Task Protocol (i.e. the columns in the various parameter tables of a Task Protocol). In addition, the interface will allow definition of Task Event Codes and Task Control Buttons.

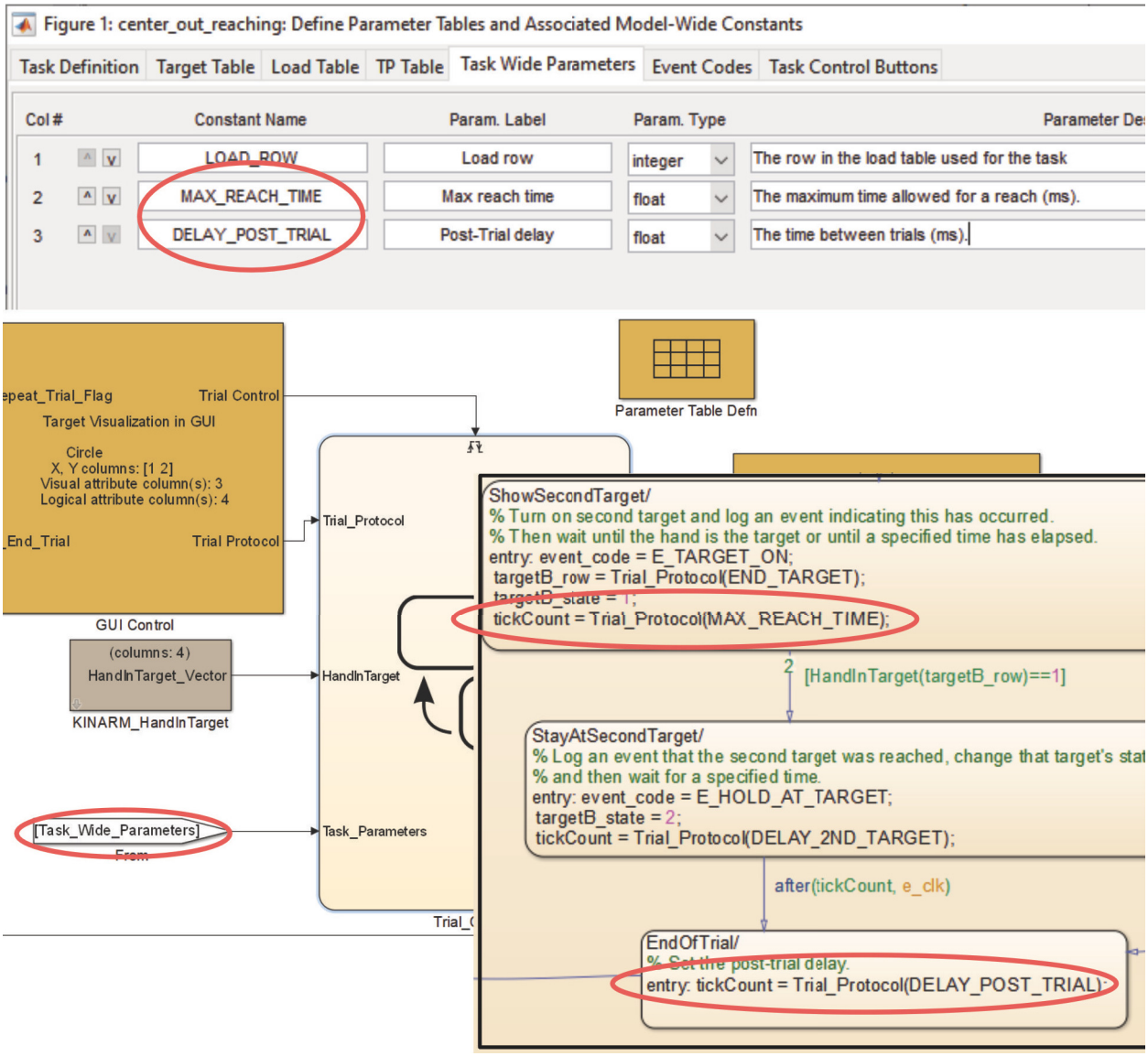
Figure 6-12: Parameter Table Defn Block



The **Task definition** tab allows you to specify default instructions for the task that will be displayed to the operator by Dexterit-E when the task runs. The **Task definition** tab also allows you to specify that the task only ever be loaded for the right or left arm, or that it can be loaded for either arm.

In the rest of the tabs of the interface it is possible to define a **Constant Name** for each parameter. These constant names become available immediately to all Simulink and Stateflow diagrams in that model once this dialog is exited (e.g., as seen in *Figure: 6-13*); these constants do not need to be defined manually in the Model Explorer (i.e. as was required for Dexterit-E 3.3 and earlier releases).

Figure 6-13: Using a Defined Constant in Stateflow



Constants defined in the `Parameter Table Defn` block can also be used directly in Simulink `Constant` blocks (*Figure: 6-14*), as block parameters (*Figure: 6-15*) and as inputs to MATLAB functions (*Figure: 6-16*).

Figure 6-14: Using a Defined Constant by Name in a Constant block

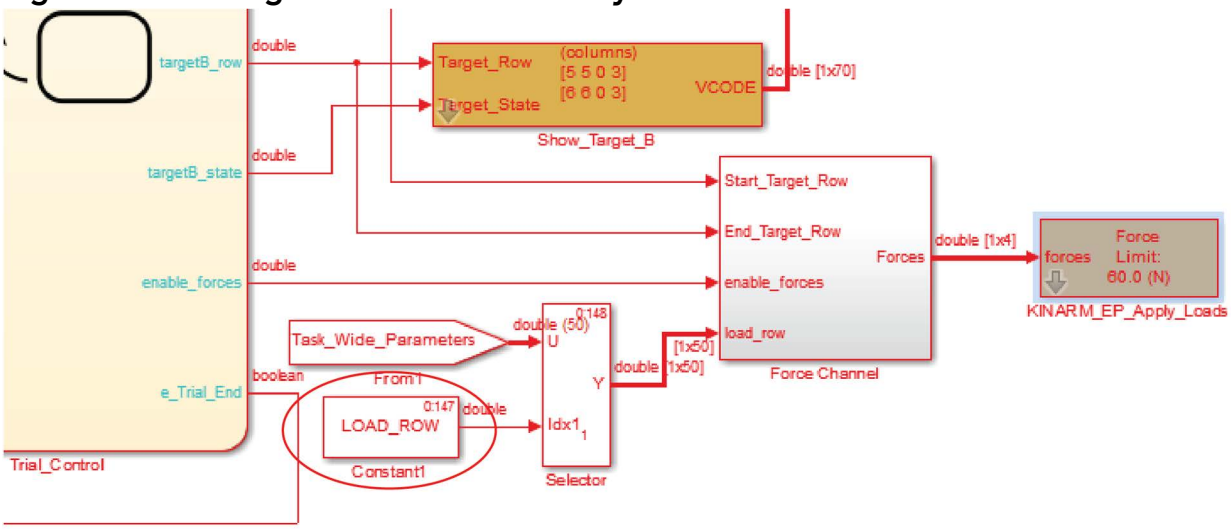
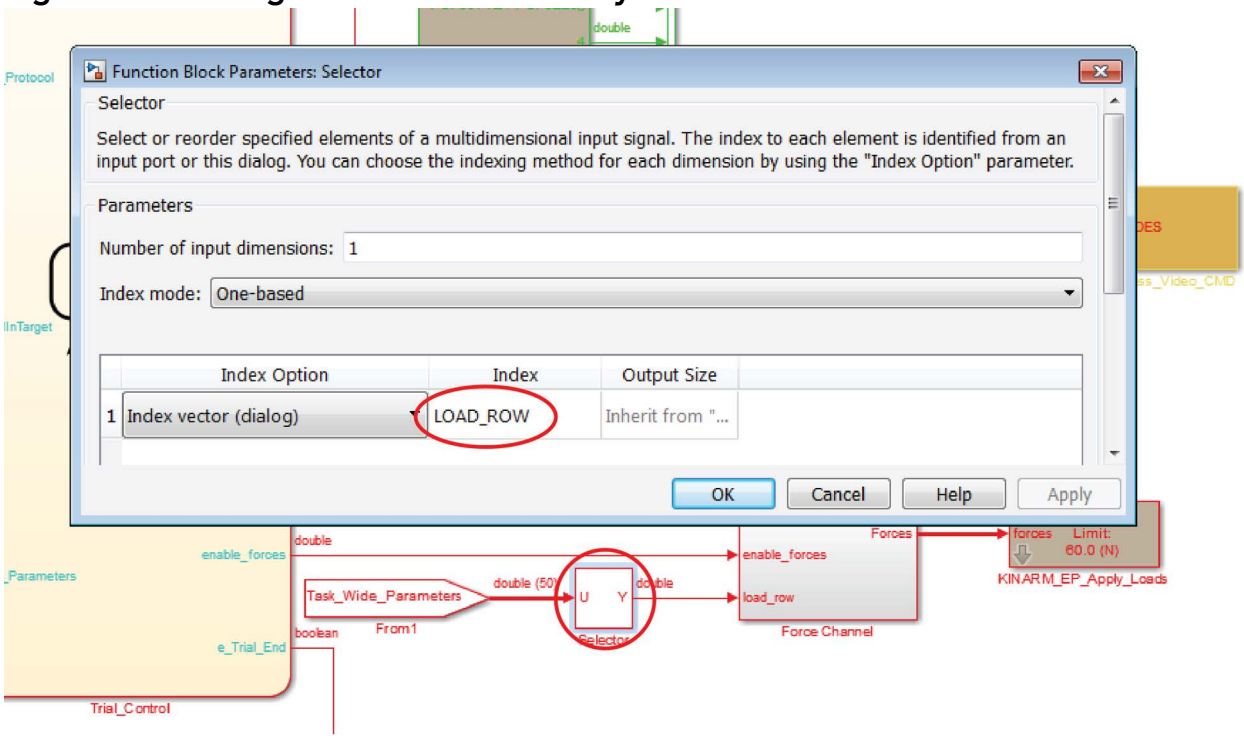


Figure 6-15: Using a Defined Constant by Name as a Block Parameter

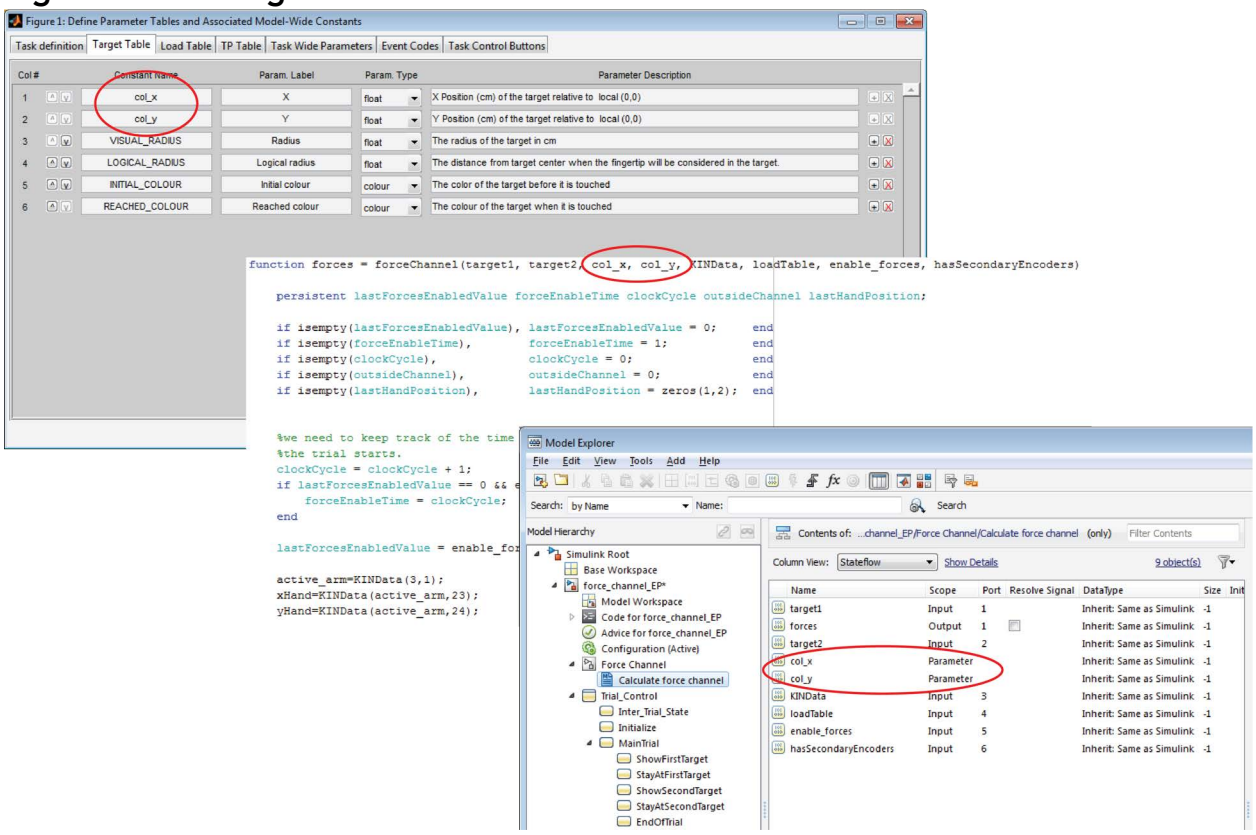


Using a constant defined in the `Parameter Table Defn` block as an input to a MATLAB function can be done in two different ways. First, one can simply use a constant block (as in *Figure: 6-14*) as an input to the MATLAB function. The second method requires a few steps as shown in *Figure: 6-16*:

1) Create an embedded MATLAB function as one normally would and define it to have an input parameter with exactly the same name as the Constant Name defined in the `Parameter Table Defn` block.
2) Save and close the MATLAB function. At this point the constant of interest will look like a normal input port on the Simulink block.
3) Open the Model Explorer, find and select this MATLAB function and change the input parameter's **Scope** to **Parameter**.

After step 3, the input port will be gone from the MATLAB function block, and the **Constant Name** defined in `Parameter Table Defn` block will be available for use in the MATLAB function as a regular constant.

Figure 6-16: Using a Defined Constant as a MATLAB Function Parameter



In the Task Wide Parameters, Target Table, Load Table, and TP Table tabs of the `Parameter Table Defn` block there are Param. Label and Parameter Description fields. The Param. Label field is displayed to the operator in Dexterit-E when editing a protocol for this model. The Parameter Description field is displayed to the operator in Dexterit-E when the given column is selected.

The Event Codes tab in the `Parameter Table Defn` block allows the definition of Task Event Codes that can be sent to the `Data Logging` block's `event_codes` input. The use of Task Event Codes is described in detail in *Section: 10 Examples of Task Program Code*.

The Task Control Buttons tab in the `Parameter Table Defn` block allows the definition of buttons that will be displayed in Dexterit-E to the operator while the task runs. The buttons in Dexterit-E will allow the operator to interact with the model during execution. The use of Task Control Buttons is described in *Section: 10.2 Task Control Buttons*.

Using the `Parameter Table Defn` block allows the end-user to safely change column orders, and add or remove columns from the various parameter table definitions. If the `Parameter Table Defn` block has different columns from what is recorded in a task protocol, the operator will be warned when Dexterit-E opens the protocol. Also, within the model, any changes to column ordering is immediately propagated through the model.

The constant names defined in your `Parameter Table Defn`block can be used in any block that references table columns, such as the various target and load blocks.

6.13 Including Pause Button Functionality

The purpose of this example is to demonstrate the Stateflow chart requirements in order for the **Pause** button in Dexterit-E to work properly. The action caused by the operator clicking **Pause** during a task depends on the choice defined in the Task Protocol (see the Dexterit-E User's Guide for more information). There are six possible pause actions defined for Dexterit-E:

- Pause immediately, continue with trial when unpaused;
- Pause immediately, restart trial when unpaused;
- Pause immediately, go to next trial when unpaused, do not repeat paused trial;
- Pause immediately, go to next trial when unpaused, repeat paused trial at end of block;
- Pause at end of trial, continue with next trial when unpaused;
- Disable pausing, the pause button is not available, to stop the task must be reset.

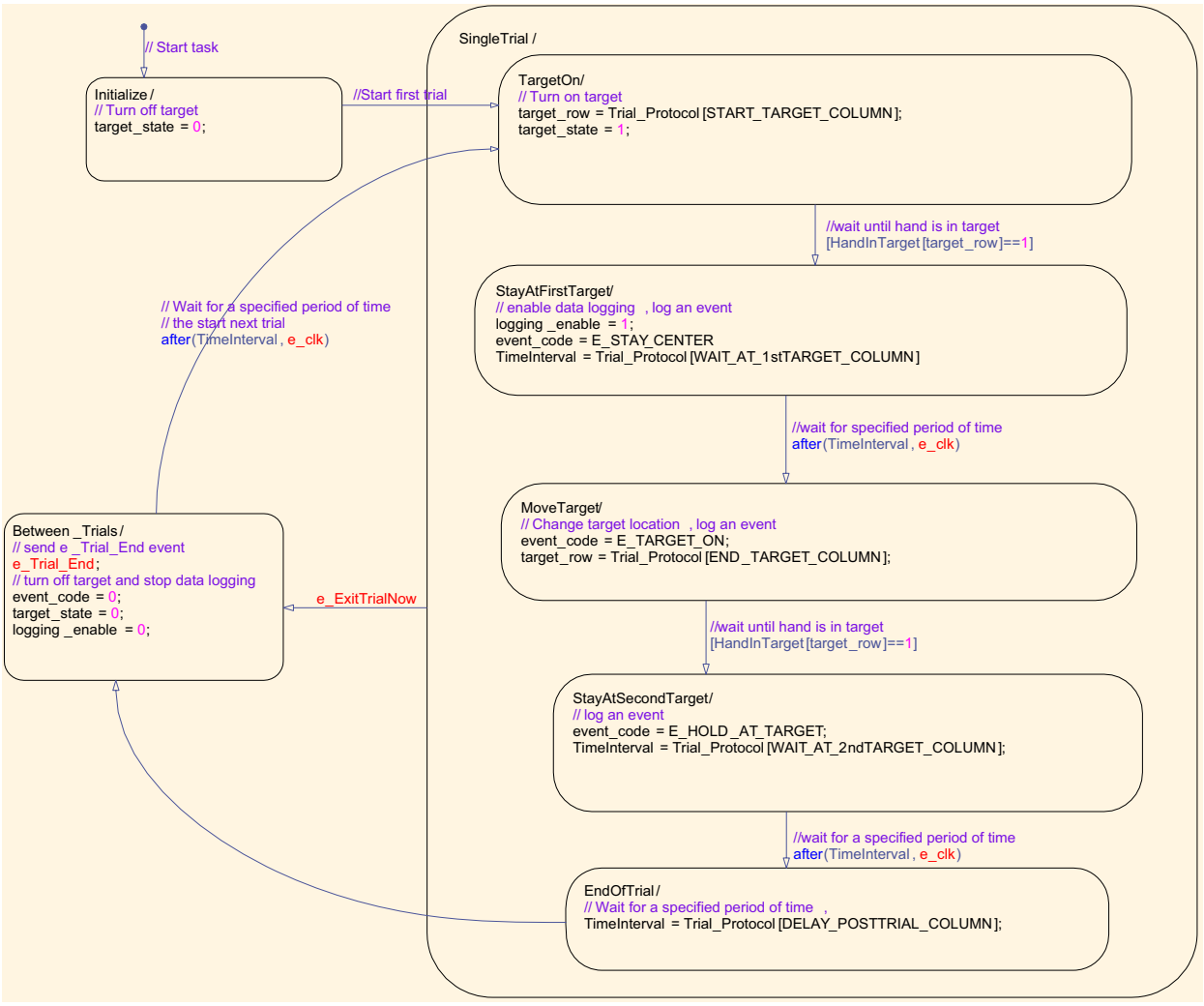
For the two pause options that pause the trial immediately and then do not complete the trial, the Stateflow chart diagram below demonstrates what code is needed in the Task Program for this functionality. If the elements described below are not present, then the Pause button will not behave as expected; instead the trial will pause immediately and then continue with the same trial when unpaused.

NOTE: A "Task Paused" Task Event Code will be stored in the output data file to record the start of the pause. All data logging halts when a task pauses.

6.13.1 Stateflow Chart for Pause Button Functionality

For **Pause** button control to work, at least two Stateflow input events must be defined for the Stateflow chart. The `Trial_Control` output of the `GUI Control` block is composed of these two Stateflow events: a 1 kHz clock and an event that can occur when the **Pause** button within Dexterit-E's GUI is clicked. In this example, the second input Stateflow event is named `e_ExitTrialNow` . This `e_ExitTrialNow` event only occurs when the chosen pause-action requires an immediate exit from the trial. For example, if the **Pause** button control in Dexterit-E is set to Pause immediately, restart trial when Unpaused, then `e_ExitTrialNow` will occur. As can be seen in the Stateflow Chart below, the occurrence of an `e_ExitTrialNow` event causes a transition to the `Between_Trials` state from any of the sub-states within the `SingleTrial` state. The key features to implement this behaviour are required: (a) a new parent-state (e.g. `SingleTrial` state) must be created which encompasses all of the other states except for the `Initialize` and `Between_Trials` states (b) a transition must be created from this new state to the `Between_Trials` state which occurs upon the second input State-flow event (i.e. `e_ExitTrialNow`).

Figure 6-17: Stateflow Chart for Including Pause Button Functionality



The significance of exiting to the `Between_Trials` state is that it contains the output Stateflow event `e_Trial_End` , which is the Stateflow event sent to the `GUI Control`block to indicate the end of trial. Also note that in many Task Programs, various actions might need to be included as part of the new exiting transition to ensure that the system exits smoothly and appropriately. For example, if large loads are being used, the implementation in this example will cause the loads to turn off immediately, which could be uncomfortable or even dangerous. Extra states and transitions to scale the load down over some given time period upon the e_ExitTrialNow event can avoid such an issue (not shown here).

NOTE: The `Between_Trials` state now has an `event_code=0` command. This command ensures that during the next trial, Task events will be registered properly even if the `e_ExitTrialNow` event occurs. Task Event Codes with a value of 0 will not be logged, nor will the operator see it listed in the Event history. See *Section: 10.1 Task Event Codes* for more information.

7 Building a Custom Task Program

Once a Task Program has been coded it needs to be compiled or "built" before it can be used with Dexterit-E. This chapter describes the basics of building a Task Program.

7.1 Build a Task Program

Building is what is done to convert the source code of a Task Program to a `<your_task>.dlm` (R2015a) or `<your_task>.mldatx` (R2019b) file, which can then be used by Dexterit-E. To build a Task Program, the `<your_task>.slx` file must be open.

There is also a script included in the TDK named `buildslrt` that will build `<your_task>.slx` from the MATLAB command prompt. The script will build `<your_task>.slx` regardless whether the model is open or not. In addition, the script will automatically clean up any intermediate build files when the build process is complete. If `buildslrt` is run with no arguments, then all SLX files in the current directory are built.

NOTE: For more information on `buildslrt`, at the MATLAB prompt type `help buildslrt`

1. Start the appropriate version of MATLAB.

   The version of MATLAB used to build the Task Program must be the same as the version of MATLAB associated with the OS running on the Robot Computer. See the Dexterit-E User Guide for how to determine, or change, which version of MATLAB was used to create the Robot Computer OS.

2. In MATLAB, switch the working directory to the directory that you want the Task Program built to.

   Typically, this directory is the same directory containing the source file (`<your_task>.slx`)

3. Start the build process using one of the following methods:
   - Open the model in Simulink, then press **Ctrl**+**B**.
   - Open the model in Simulink, then click the **Build** button in the Simulink toolbar.
   - Open the model in Simulink, then choose **Code** -> **C/C++ Code** -> **Build Model** (MATLAB R2015a only).
   - At the MATLAB command prompt, type `buildslrt ('<your_task>.slx'` to build a single model.

NOTE: type `buildslrt` to build all models in the working directory.

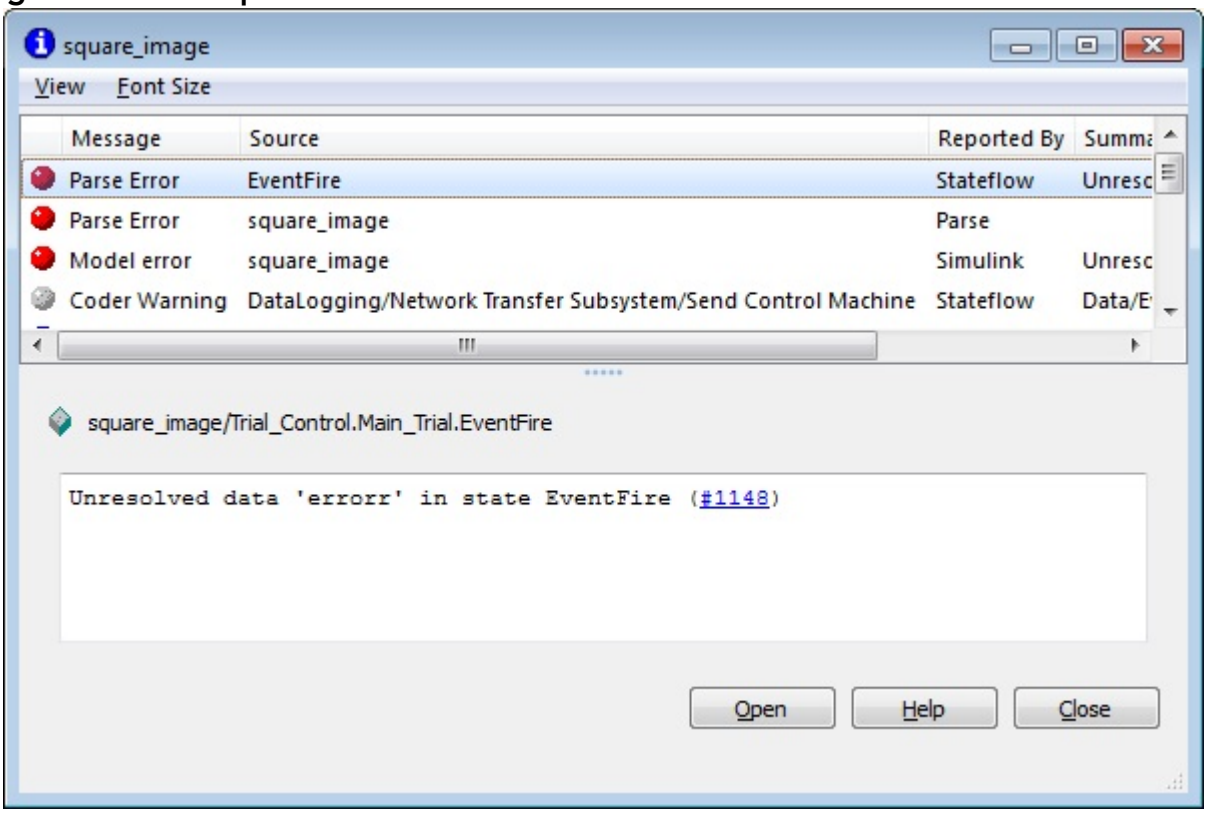
Any errors that occur will show up either in the MATLAB workspace Command Window or in a new window.

If the build process is successful, it produces a `<your_task>.dlm` (R2015a) or `<your_task>.mldatx` (R2019b) file in the working directory. You can then make this file available to Dexterit-E.

7.2 Common Errors During the Build Process

When an error occurs during the build process, the most common result is the creation a new window which lists the status and results of the build process, including the errors, an example of which is shown in *Figure: 7-1*. The following are some tips that can be useful in fixing common build errors.

Figure 7-1: Example Build Process Error



7.2.1 General Tips for Resolving Build Errors

- Occasionally, a long list of errors will be shown in the popup window following an attempt to build a new Task Program. Often, many of the errors will simply occur as a result of the first error, so a good strategy to try is to fix the first error and then rebuild the Task Program before attempting to fix the subsequent errors.
- Occasionally, it will not be obvious from the popup window what the cause of the error is. In this case, updating the Task Program (rather than rebuilding it) can produce a different, more meaningful error message. To update a Task Program, right-click on the Simulink workspace of the Task Program and select `Update Diagram`.
- Rarely, a Task Program will continue to fail to build properly, even after all apparent errors have been corrected. In this case, all of the temporary, intermediate files created by the build process should be deleted. To accomplish this, at the MATLAB prompt run "`cleanslrt`" to delete all intermediate build files and close MATLAB.Restart MATLAB and rebuild your Task Program.

7.2.2 Specific Build Errors

- "Error using ==> xpc\private\xpcload Could not find target". If the preceding Build Error message appears in the popup window, then you need to uncheck an option within your Task Program. Open your Task Program, select **Simulation** -> **Model Configuration Parameters**, and in the left pane of the *Configuration Param-*

eters dialog select **Code Generation** -> **Simulink Real-Time Options**. Uncheck **Automatically download application after building**. Rebuild your Task Program.

NOTE: This error should not occur with R2019b as it does not have this setting option

- "`Unresolved data 'RAND_MAX in ...`". The RAND_MAX value is defined in the `stdlib.h` file which must be included as part of the custom code library for the Stateflow chart in which the constant is called. See *Section: 10.18 Random Numbers in Stateflow* for more details.

8 Using and Testing a New Task Program

To use a custom Task Program, it must be made available to Dexterit-E. This chapter introduces the issues associated with making a Task Program available to Dexterit-E and initial testing of the Task Program.

WARNING: When using an Kinarm Exoskeleton Lab, if your task commands forces it is critical to maintain physical control of the robot arms either by holding the arms or by having a subject place their arms in the troughs. Commanding a load in an unloaded robot can lead to very high speeds, which can produce very high contact forces when they contact mechanical stops. Not only could this lead to damage of the robot, but it could lead to subject or operator injury.

8.1 Make Your Task Program Available to Dexterit-E

To make the new Task Program available to Dexterit-E, the following steps must be completed:

1. Ensure that the newly-created `<your_task>.dlm` or `<your_task>.mldatx` file is in a sub-directory in the Task Programs directory on the Dexterit-E Computer (i.e. "…\My Documents\Dexterit-E x.x Tasks\<your task>\").
   - If this is a new Task Program, create a new sub-directory in the Task Programs directory on the Dexterit-E Computer (i.e. "…\My Documents\Dexterit-E x.x Tasks\<new task>").
   - If this is an update to an existing Task Program, then replace the previous version of `<your_task>.dlm` or `<your_Task>.mldatx` with the updated version.

   NOTE: Dexterit-E will only recognize the first Task Program it finds in a sub-directory, so if multiple `.dlm` or `.mldatx` files are found in the same sub-directory, the others should have their extensions re-named, e.g. to `<your_task>.OLD`

2. Ensure that there is a Task Protocol available:
   - If this a new Task Program, you can either:
   – create a new Task Protocol from scratch within Dexterit-E (see Dexterit-E User Guide for details).

  - copy over a similar Task Protocol from another Task Program and then modify it within Dexterit-E.
- If this is an update to a Task Program, then the existing Task Protocols will be automatically available. However, they may require edits depending on the changes to the Task Program.

8.2 Error - "CPU Overload"

If Dexterit-E reports that your task had a "CPU Overload", then your Task Program running on the Robot Computer has taken too long to execute during a single clock cycle (250 µs). There are multiple causes/fixes for this error (please search MathWorks' documentation for CPU Overload). You can use the following tools to help diagnose the problem:

- Add the `Task Execution Time` Simulink block to your model and set the step time to `BKIN_TIME_STEP`. Send the output from this block to a graphical `SLRT Scope`. Set the Y-Limits on the Scope to `[0, BKIN_STEP_TIME]`. When you run your task this scope will display graphically on the Robot Computer screen the amount of time your task takes to run. Spikes in time that are close to the upper bound of the graph indicate the potential for a CPU Overload.
- Removing Simulink blocks from your model can help you pinpoint code which uses too much CPU time. You can right-click on Simulink blocks and select **Comment Out** to temporarily remove a block from your model. Then recompile and run your task. As you do this you should see changes in the graphed Task Execution time. This should help find areas in your code that use the CPU heavily.

Below is a list of some causes and solutions to CPU Overload errors (please search MathWorks' documentation for additional information):

- Some very CPU intensive operations have been added (and will need to be rewritten to be more efficient). In particular, accessing and manipulating large matrices can be quite expensive. It may be necessary to
  - Optimize matrix calculations to use column major indexing.
  - Set the step time for the block to something longer than 250 µs
  - Find more efficient algorithms
- Copying large matrices between blocks can be expensive. Depending on the code generation, a large matrix may be copied rather than simply passed between 2 Simulink blocks. This problem mainly occurs when the matrix is defined in a Constant block and can be corrected by placing the Constant block in an enabled sub-system and disabling the sub-system. This signals to Simulink that the parameter will not be tuned.
- Too fast a sampling rate of the Task Program - the default execution rate for Task Programs is 4 kHz. To set the execution rate, from within your Task Program in Simulink, choose **Simulation** -> **Configuration Parameters**. In the *Configuration Parameters* window choose **Solver** on the left-hand pane and look at **Fixed-step size**

(fundamental sample time). The default value is BKIN_STEP_TIME (0.00025 s). If you have increased the rate beyond the default (e.g. to 8 kHz), then this could be the source of the problem.

- Too many Analog Input (AI) channels or too long of a sampling time (a.k.a. scan interval) for those AI channels - data acquisition of analog input channels can take a significant time because each channel must be sampled in succession. The amount of time taken by analog input data acquisition is equal to the number of channels of analog input multiplied by the sampling time or scan rate for each channel. For example, if the per-channel conversion time is 12.8 µs and 32 channels are being recorded, the total acquisition time is 12.8 µs * 32 = 410 µs, which is too long for a 4 kHz update rate, and even if the AI channels are running at a slower 2 kHz execution rate that leaves only 90 µs. Try reducing the number of AI channels or sampling them at a slower rate to determine if this is the source of your problem.

  NOTE: We recommend using the `Analog Inputs` block from the Kinarm I/O library. This is a drop in replacement for the PCI-6071E and PCI-6229 analog input blocks, but with some enhancements. This block makes it possible to move models without recompiling between older and newer Kinarm Labs which have different analog input cards. Once the new block is added, double-click it to edit the scan intervals for the different cards.

9 Test and Debug a Task Program

Once a Task Program has been successfully built, it must be tested using Dexterit-E. Task Programs cannot be tested within Simulink (i.e. you cannot simulate a running of a task within Simulink). This chapter provides some tips that can assist with this process.

9.1 Second Mouse

If you are testing by yourself it can be difficult to calibrate the Kinarm Exoskeleton Classic Lab if you cannot reach the Dexterit-E Computer from the Kinarm Lab. In order to make this easier you can acquire a second mouse (possibly with an USB extension cable) and plug it into the Dexterit-E Computer. Placing a piece of opaque tape over the sensor of the second mouse will ensure that only its buttons work. With this set-up you can use the main GUI mouse to place the cursor, then carry the second mouse with you to the Kinarm Exoskeleton Classic Lab and use it to click through the calibration steps.

9.2 Debugging with SLRT Scopes

During the debugging phase of testing a Task Program, it is often useful to utilize SLRT scopes. SLRT scopes allow you to view Simulink signals numerically or graphically in real-time on display connected to the Robot Computer. For example, if a load does not feel correct, you can view the values of the commanded forces or torques being applied, as well as the signals being used to create those force or torque commands. See *Section: 10.6 SLRT Scopes*

9.3 Disabling Loads

During the debugging phase of testing a Task Program, there exists the possibility of accidentally creating dangerous loading conditions. To avoid this problem, we recommend the following:

- Add SLRT Scopes to the Task Program to view the loads,
- Disable any motors on the system being controlled using the Emergency Stop button,
- Run the task with the motors disabled and the commanded loads on the SLRT Scopes.

For more information on SLRT scopes, see *Section: 9.2 Debugging with SLRT Scopes*.

9.4 Separation of Stateflow and Simulink

The Stateflow code controls the logical flow of behaviour during a task, whereas the Simulink code is more related to inputs and outputs that interact with a subject. It is useful to attempt to focus on these aspects of the code independently where possible. For example, if your task includes creating loads on your Kinarm robot, by disabling the Kinarm robots you can safely test and debug the behavioural logic in your Stateflow chart without having to worry about unsafe loading conditions.

9.5 Simplify the Task Program

If a new Simulink block has been created for your Task Program and it will be used multiple times in the Task Program, if possible it is best to have only one instance of the block in your task until it has been verified that the block works exactly as desired. This approach will not only simplify the debugging phase, but will also help ensure that incorrect, bug-filled copies of the block are not accidentally used.

9.6 Verifying Loading Conditions

If a Task Program produces loads, the first test of correct load implementation in a Task Program is often based on perception of how the loads feel. Although this is a good first test, a few issues need to be considered when trying to interpret the perception of those loads.

- Is the testing being carried out at the point that the loads are being applied? For example, if the loads are defined for the Kinarm robot at the 'fingertip', is the testing being done at that fingertip? If not, then a discrepancy will exist.
- Has testing being carried out in the absence of loads to notice the effects of robot inertia? The Kinarm robot's inertia ellipse is aligned with that of the arm (i.e. the shoulder joint is 'heavier' than the elbow joint). If the robot is being held at the end-point, the effects of this inertia ellipse can be felt in two ways. The first is the direct effect of the inertia on force (higher forces will be required to move the Kinarm robot along the major axis of inertia versus the second). The second is an indirect effect of the inertia - unless carefully controlled, movements along the major axis of inertia will be slower, and if the load is a velocity-dependent load, the result will be less of a load.
- For the Kinarm robots, the loads commanded by a Task Program are applied to the robot, which in turn applies the loads to the subject. Under static conditions these loads are identical, but under dynamic conditions, the mass and inertia of the robot can change the loads slightly due to Newton's equations of motion (i.e. F = ma). There are two methods of examining this issue. The first is to perform the same experiment in the absence of loading conditions to 'feel' the effects of moving the Kinarm robot in different directions and then compare that to the loaded conditions. The second approach is to determine the effects analytically, which requires

one to implement the equations of motion and calculate the effects of the Kinarm robot.

9.7 Use Task Event Codes to Monitor Stateflow

In your Stateflow chart it can be useful to add extra Task Event Codes which are passed to the `DataLogging` block. The extra Task Event Codes can help you better verify that your Stateflow chart is moving through the various states in the manner that you expect it to be.

NOTE: Having many Task Event Codes in your Stateflow helps both with debugging and later analysis of collected data.

9.8 Make Changes in Small Steps

In any experiment, changing too many things at once throws into question what caused things to happen. This is no different when building task programs. Try to make only small changes to your program at any time, that way you can narrow down errors much more easily and correct them immediately.

WARNING: When testing a new or newly-edited Task Program, take appropriate safety precautions. This includes having the Kinarm Lab's Emergency Stop button readily accessible and keeping a firm grip on the robot being controlled. Mistakes in coding can result in potentially dangerous behaviour of the system, resulting in either damage to the system and/or injury to a subject and/or operator.

10 Examples of Task Program Code

This chapter provides numerous examples of code to highlight features common to many Task Programs. Most of these examples build upon the basic example shown in *Section: 6.7 Centre-out Reaching Task*.

NOTE: Many of the examples provided here use MATLAB as the Action Language in Stateflow. This means that any code written in the Stateflow chart uses MATLAB syntax. In previous versions of this guide, the examples used C as the Action Language for Stateflow. For details on converting your Stateflow to use MATLAB as the Action Language see *Section: 14.17 Update the Stateflow Action Language to MATLAB*.

10.1 Task Event Codes

Task Event Codes provide a method of time-stamping when something occurred and what that something was, which can be useful for subsequent data analysis. Examples of typical Task Event Codes that are time-stamped in this manner include target on/off events and user-defined errors (e.g. the subject moved too slowly), however, they can be anything that the end-user wishes. Task Event Codes are end-user specific; they have no meaning to Dexterit-E. Task Event Codes are displayed in the Dexterit-E GUI as they occur during run-time and are automatically saved to the exam file.

To save a Task Event Code, a Task Event Code must be passed into the `DataLogging` block in the Simulink code (see example below). Task Event Codes are logged only when the `event_codes` input to the `DataLogging` block changes. Within a Simulink model, Task Event Codes are treated and evaluated as numbers. However, names and descriptions can be defined for Task Event Codes in the `Parameter Table Defn` block (see example below), such that if a name is defined for a Task Event Code, then when that Task Event Code occurs and gets logged, then the name of the Task Event Code gets saved to the exam file. If there is no name defined for a logged Task Event Code, then a name is auto-generated based on the Task Event Code's number (see below).

Some rules for Task Event Codes are:

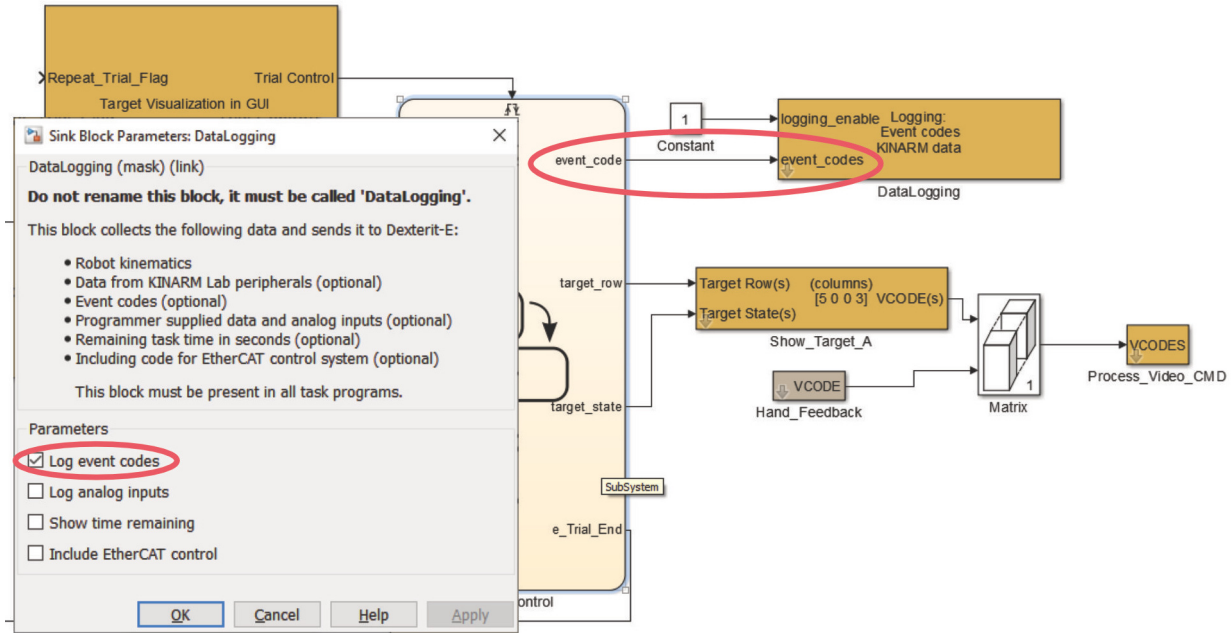
1) In the Dexterit-E user interface, and in the data files generated by Dexterit-E after running a task, Task Event Codes are saved as, and referred to as, events. (Note: Task Event Codes should not be confused with Stateflow Events, which do not appear in the Dexterit-E user interface, and which are not saved to data files).
2) When a Task Event Code is saved in a data file it is saved (as an "event") with the name from the `Parameter Table Defn` block, if one exists. If a Task Event Code is sent that is not defined by name in the `Parameter Table Defn` block then it will be saved with the name "Unnamed Event: code [value]".
3) In addition to the name, the number of the Task Event Code is also saved to the data file.

4) The `Parameter Table Defn` block will allow you to define a maximum of 511 named Task Event Codes codes with values between 1-65535.
5) If a Task Event Code is set to zero then no event is recorded. In the `Parameter Table Defn` block E_NO_EVENT is a constant created by default and set to zero.
6) Task Event Code values can be in the range 1-16.7 million. Negative values are converted and saved as the absolute value of the Task Event Code.
7) Task Event Codes are only recorded by Dexterit-E when the `event_codes` input on the `Data Logging` block changes. In order to record the same Task Event Code twice in a row you can send the negative of the Task Event Code. For example, if you sent Task Event Code 5, then later Task Event Code -5, they will both be registered as Task Event Code 5, but at different times. It is also possible to set the `event_codes` value to E_NO_EVENT in between setting `event_codes` to a required Task Event Code value.
8) Task Event Codes are recorded at 1 kHz. If the value being passed into the `Data Logging` block for a Task Event Code changes more than once during a 1 ms time interval then only the last value of the Task Event Code is recorded. If you need to be able to register multiple Task Event Codes during 1 ms then a vector of Task Event Codes can be passed to the `Data Logging` block. Each element of the vector will treated independently (i.e. if element 1 changes at a given time frame, but not element 2, then only the Task Event Code for element 1 will be logged).

10.1.1 Simulink Code for Task Event Codes Example

This Task Program is the same as *Section: 6.7 Centre-out Reaching Task*, but with a new output added to the Stateflow chart called `event_code`. This new output is connected to the `event_codes` input of the `DataLogging` block, which has been made available by checking Log event codes in the `DataLogging` block's dialog. It allows Task Event Codes to be recorded during a task.

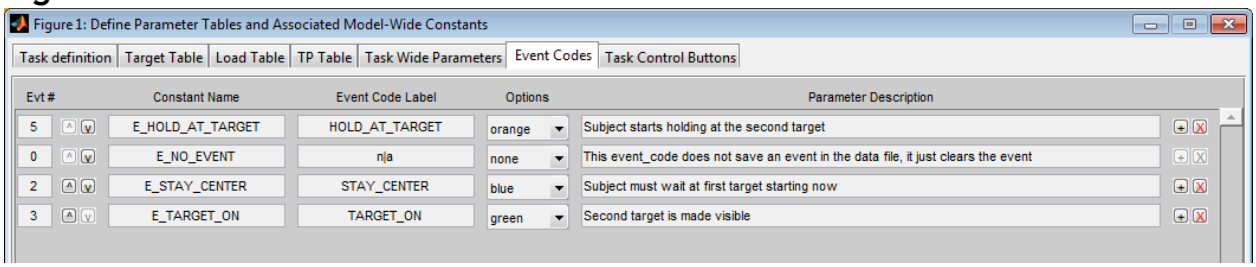
Figure 10-1: Simulink Code for Task Event Codes Example



10.1.2 Parameter Table Defn Block for Task Event Codes Example

To save the name and description of Task Event Codes, the end-user must define the Task Event Codes in the `Parameter Table Defn` block.

Figure 10-2: Task Event Codes in the `Parameter Table Defn` Block.



- **Evt #** – is value of the Task Event Code used in Simulink (i.e. when the Task Program logs an `event_code` of this value, then the name and description associated with that code as defined here is what gets saved in the data file as an "event")
- **Constant Name** - A name that can be used within any Stateflow code, or in Simulink blocks as described in *Section: 6.12 Using the Parameter Table Defn Block*.
- **Event Code Label** - The name that will be saved for the Task Event Code in the data files. Reserved labels that are not allowed are listed in *Section: 14.13 System Generated Task Events Codes*.
- **Options** - The available options are:
  - **None** - No extra handling is done.
  - **Error** - If this Task Event Code is detected then the trial is marked as an Error Trial (*Section: 10.13 Error Trials: Repeating and/or Reporting*).

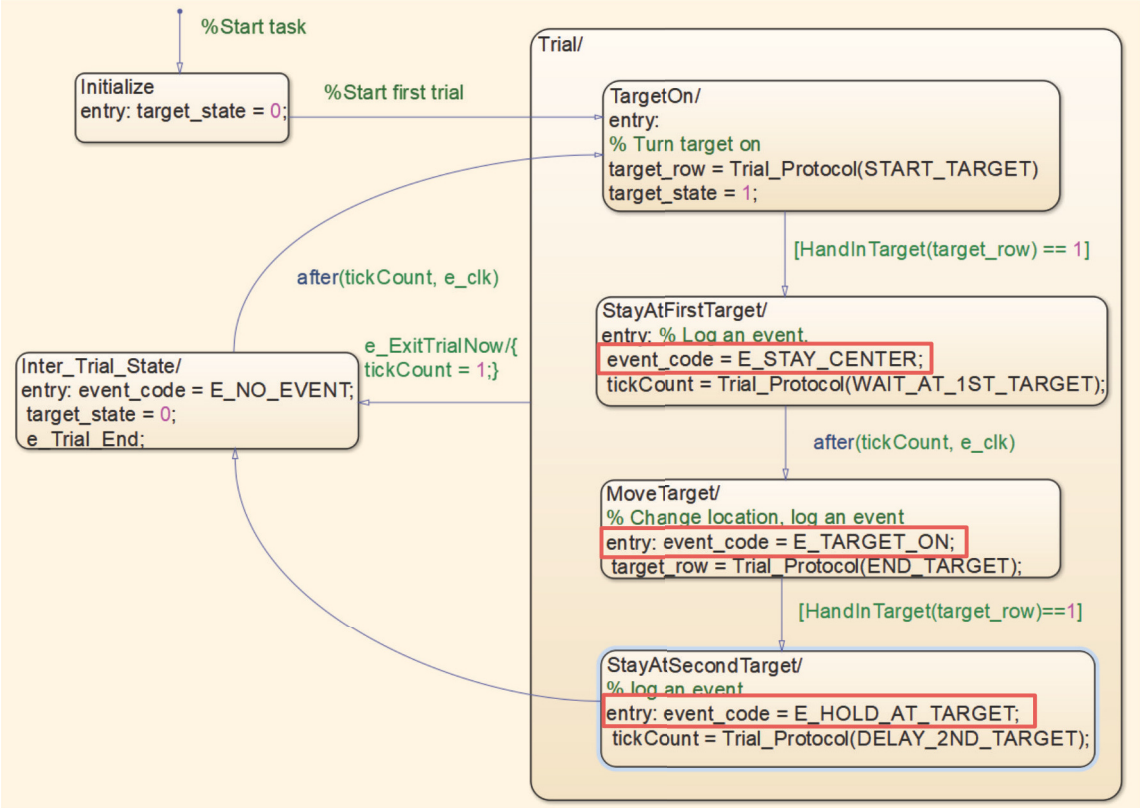
  - Hidden - The Task Event Code is not displayed to the operator during a task in Dexterit-E user interface (unless **Show hidden events** is selected from the menu).
  - **Red**, **Green**, **Blue**, **Orange**, **Purple** - The given colour is used when displaying the Task Event Code in the Dexterit-E user interface.

- **Parameter Description** – A textual description of the meaning of this Task Event Code.

10.1.3 Stateflow Chart for Task Event Codes Example

In the Stateflow Chart shown next, the output `event_code` is assigned various values when different states are entered. The values that `event_code` is set to in this example (e.g. `STAY_CENTER`) are constants, defined in the `Parameter Table Defn` block.

Figure 10-3: Stateflow Chart for Task Event Codes Example

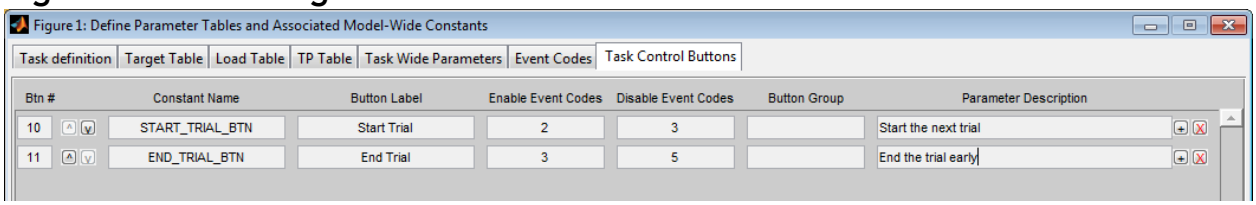


Task Event Codes are only registered and saved when the `event_codes` input to the `DataLogging` block changes. It is therefore imperative that the `event_code` be forced to change at least once per trial. In this Stateflow Chart example, there is only one branch to the flow chart, and so the `event_code` will cycle through different values and will always be set appropriately. However, for a Stateflow Chart with multiple branches, setting `event_code=E_NO_EVENT` as part of the `Between_Trials` state is required. See *Section: 6.13 Including Pause Button Functionality* for an example of this type.

10.2 Task Control Buttons

For some tasks it is useful to have the operator able to provide input in order to mark events or advance trials. A task can define buttons that are displayed in Dexterit-E. The buttons are defined within the `Parameter Table Defn` block. Button clicks during task execution are recorded automatically in the data files as Task Event Codes with the name "TASK_BUTTON_[id]_CLICKED". Here is an example of defining buttons (assuming the Task Event Code definitions in the *Section: 10.1.2 Parameter Table Defn Block for Task Event Codes Example*):

Figure 10-4: Defining Task Control buttons.



The variables in the interface have the following meanings:

- **Btn#** – This is an identifier for the button. If buttons are not given group and layout information they are placed in the GUI in order of ascending index. Button numbers may be any value from 2 to 254.
- **Constant Name** - A name that can be used within any Stateflow code, or in Simulink blocks as described in Section 3.12.
- **Button Label** – The text that will appear on the button in the GUI. This is also the string used in both Dexterit-E and data files to record and identify the Task Event Code when the button is clicked.
- **Enable Event Codes** – A comma separated list of defined Task Event Codes that will cause the button to enable. If the field is blank then the button is enabled by default.
- **Disable Event Codes** – A comma separated list of defined Task Event Codes that will cause the button to disable.
- **Button Group** - Allows the grouping and laying out of buttons (see description below).
- **Parameter Description** – The text that will appear beside the button in the GUI.

In order to make use of clicks of your defined buttons you will need to use a `From` tag in your Simulink code. In the `From` tag dialog, select `task_control_button`. When the operator clicks a Task Control Button in the Dexterit-E user interface, the `From` tag in Simulink will output the value of the button # for 1 ms. The rest of the time the `From` tag will output a zero.

Below is an example of a simple button which when clicked will produce a Simulink event that can drive a Stateflow chart (e.g. to advance to the next trial within the Stateflow chart). In the Simulink code the button click is used to drive a Stateflow event. Because the `task_control_button From` tag holds its value for 1 ms it can also be used as a normal data input to the Stateflow chart.

Figure 10-5: Example Task Control Button in the GUI.

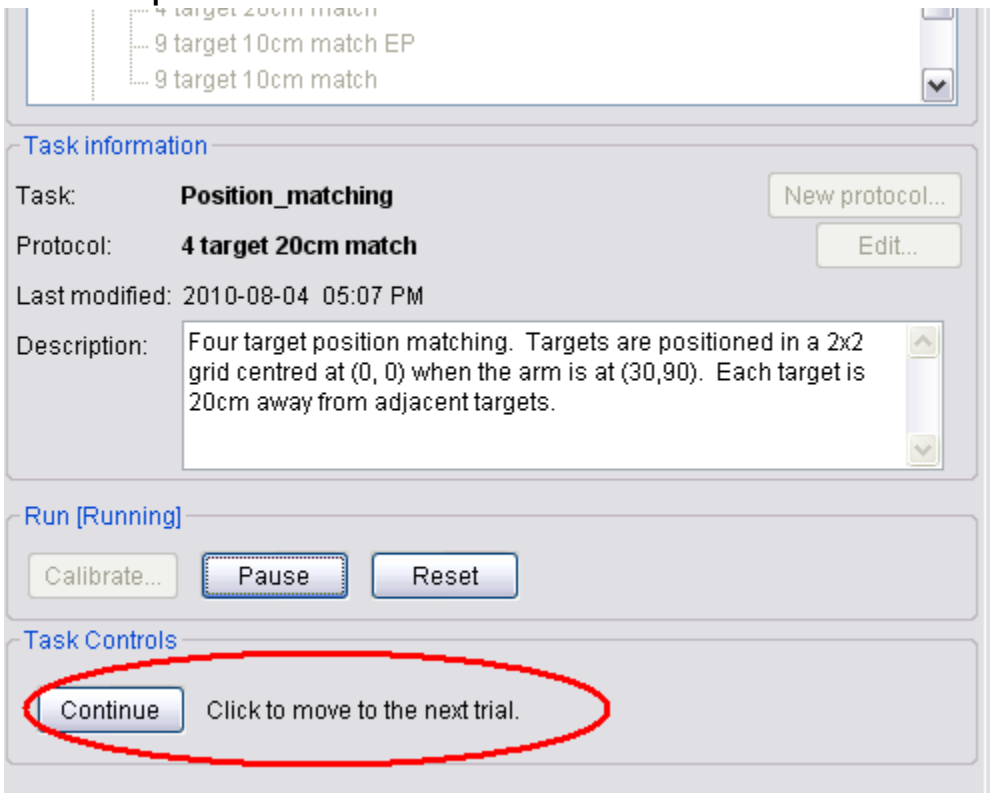
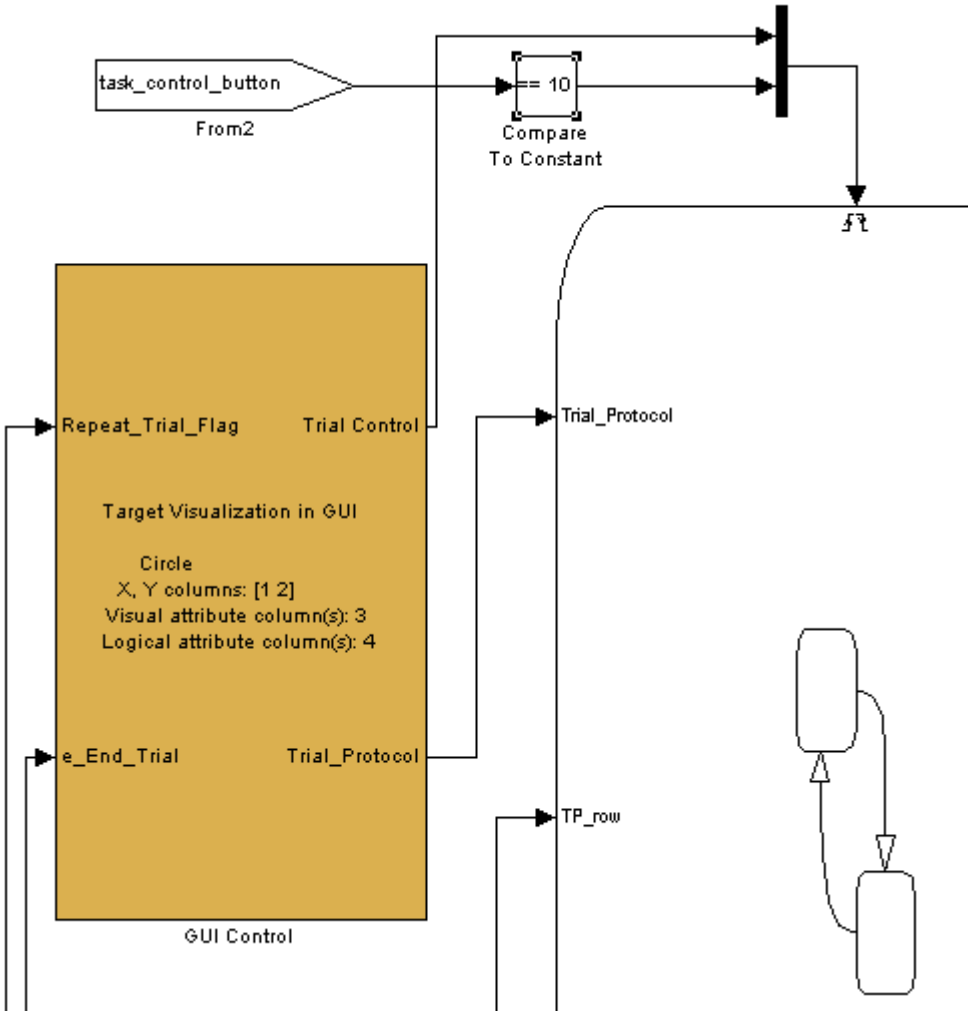


Figure 10-6: Using a Task Control Button in Simulink



10.2.1 Grouping Task Control Buttons in Dexterit-E

You can group several Task Control Buttons together using the **Button Group** column in the *Task Control* buttons tab of the `Parameter Table Defn` block. The format of the button group string is:

[Group ID],[rows]x[cols],[position]

Where:

- **Group ID** - any integer. All buttons defined within the same group ID are considered part of the same group.
- **rows** - the number of rows in the grid that is used to lay out the group of buttons.
- **cols** - the number of columns in the grid that is used to lay out the group of buttons.
- **position** - the position in the grid where the button should be placed. 1 is the top left corner.

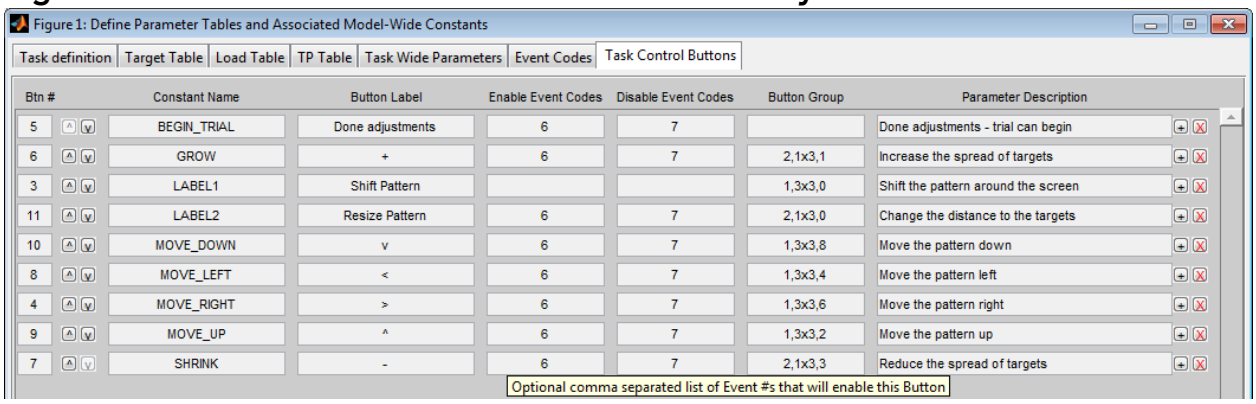
For example, if the rows and columns are defined as 3x4, the the possible positions on the grid are:

Table 10-1: Example Button Layout

1	2	3	4
5	6	7	8
9	10	11	12

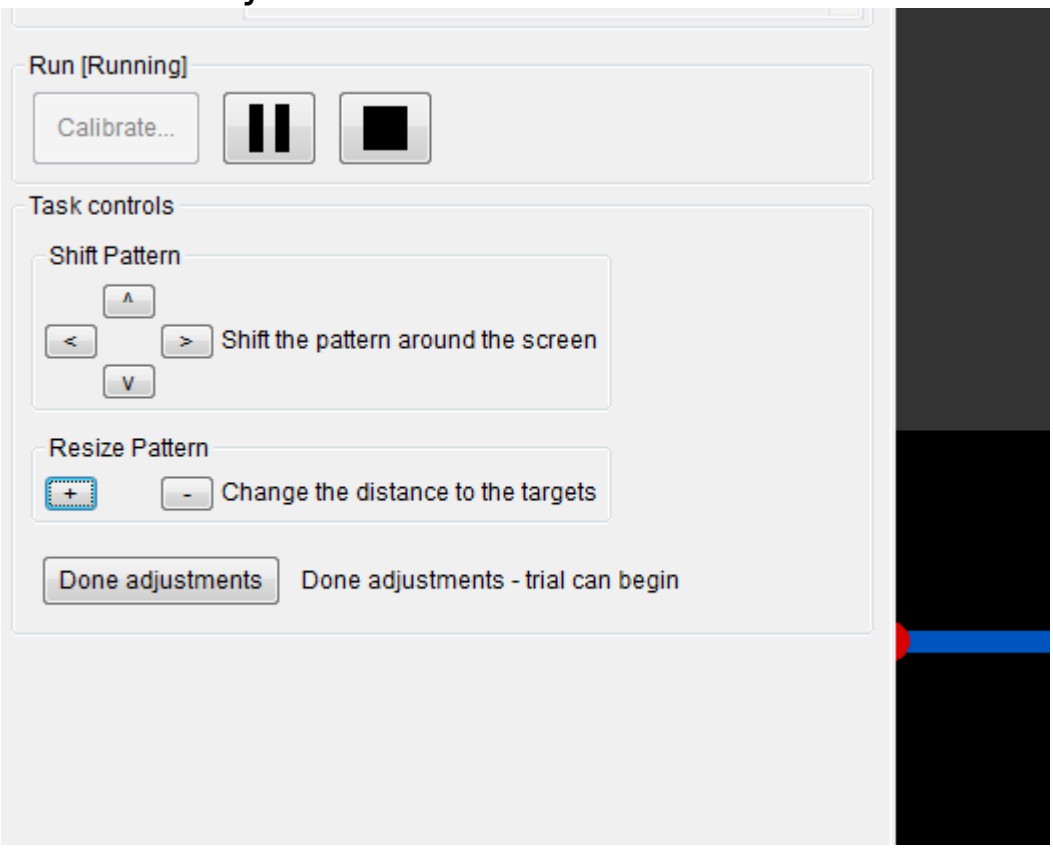
Buttons defined at position 0 become the optional label for the group of buttons. *Figure: 10-7* and *Figure: 10-8* show an example of a more complex button layout. This example includes two button groups and one solitary button. If no button is defined at a position then that position is considered empty and is drawn in the normal background colour of Dexterit-E.

Figure 10-7: Parameter Table Definition of Button Layout



Based on the layout defined above, the buttons would appear in Dexterit-E as:

Figure 10-8: Button Layout in Dexterit-E



10.3 Analog Inputs

The Analog Inputs option in the `DataLogging` block allows a Task Program to save data recorded from an analog input device synchronously with other automatically record information from the Kinarm robots. Examples of data that might be recorded include electromyograms, torque outputs from the motor's servo amps, or even non-analog signals such as custom data streams derived internally to the Task Program.

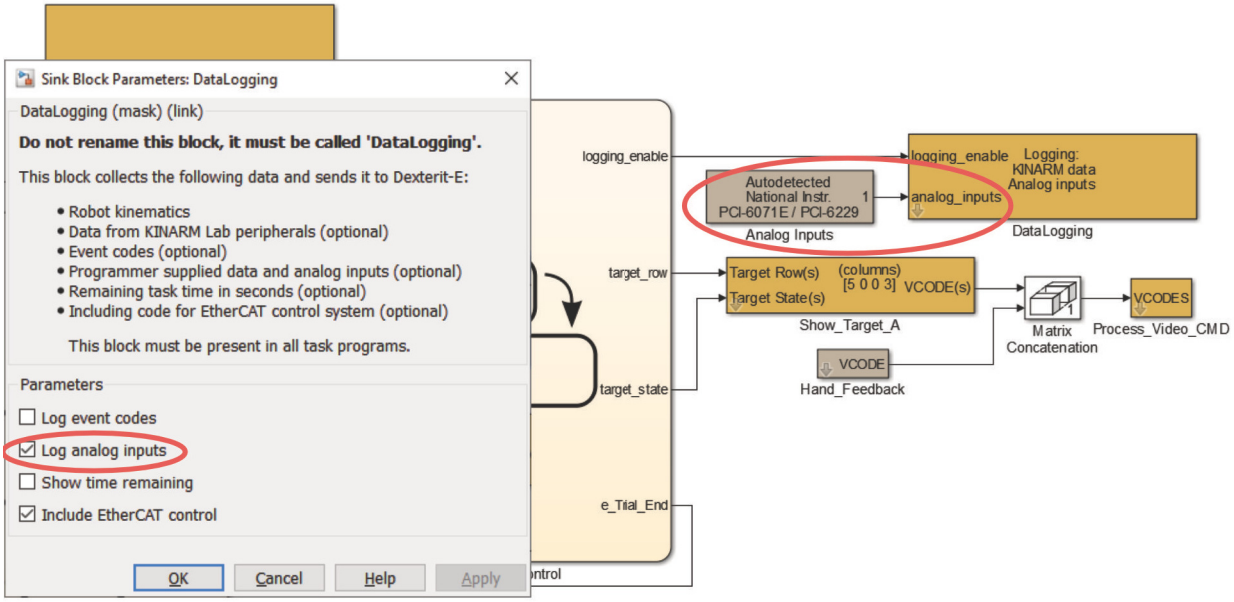
NOTE: Recording of analog inputs requires an analog input card supplied with a Kinarm Lab.

10.3.1 Simulink Code for Analog Inputs Example

In this example, the `Analog Inputs` option in the `DataLogging` block has been checked, which causes an input port to appear in the `DataLogging` block. The `Analog Inputs` block from the Kinarm I/O library of the TDK is added to the Simulink diagram and then configured and wired into this new input on the `DataLogging` block. The Analog Inputs block from the TDK is able to detect which type of analog input card you have and, therefore, acquire data appropriately.

This means that tasks can be built without needing to know which card you have. As with data logging, the analog data are only recorded when the `logging_enable` input of the `DataLogging` block is equal to 1 (True).

Figure 10-9: Simulink Code for Analog Inputs Example



NOTE: The `Analog Inputs` block in the Simulink diagram must be present to record the actual torque values of the motors for Kinarm Classic Labs if desired. By default, only the commanded torques are recorded as part of the Kinarm data.

NOTE: Kinarm Labs purchased in 2011 or later should use a default of 16 or 32 channels. The Task Protocol manages the channels that are saved. Older systems that have not replaced or updated their Robot Computer (e.g. with PN 11103 or greater) may have trouble dealing with this many default channels.

10.3.2 NI PCI-6071E or PCI-6229 Data Acquisition Cards

Human Kinarm Labs optionally include a National Instruments (NI) PCI-6071E or PCI-6229 card. If you plan to collect the data from the analog channels on the card you are encouraged to use the `Analog Inputs` block in the Kinarm I/O library. This block allows you to control the sampling time (i.e. scan interval) for the analog-to-digital acquisition, and will make your task portable between older and newer Kinarm models. Reasons for controlling the scan interval are to ensure that you do not get cross-talk between channels (if the scan interval is too short), and avoiding CPU overload (if the scan interval is too long). See *Section: 8.2 Error - "CPU Overload"* for more information related to this block.

10.4 Saving Custom Data

The `analog_inputs` input on the `DataLogging` block can also be used to save any custom data. This input accepts a vector of doubles. Each element of the vector represents a separate channel of data that will be optionally saved in your data file. Several things are important to know about the saving process:

- In order for the data to be saved with your exam you will need to name the channel and mark it as one you want to save when you create or edit your Task Protocol in Dexterit-E. For details on protocol editing please see the Dexterit-E User Guide.
- Data is always saved in the data files as single precision. In order to save integer data, you will need to convert the integers to doubles prior to connecting them to the `analog_inputs` input on the `DataLogging` block. During analysis, those values can then be converted back to integers in MATLAB as necessary.
- Your model will be running at 4 kHz, but the data will only be streamed to Dexterit-E at 1kHz. If you need to actually save data at 4 kHz then you will need to send 4 channels of data (with appropriate time-delays) and then reconstruct them later in MATLAB during analysis.
- A maximum vector size of 100 elements can be sent to the `analog_inputs` input. This means you can save a maximum of 100 channels of custom data. Creating more than 100 channels of custom data may lead to your exam files being corrupted.

NOTE: You do not need to save any kinematics created by your Kinarm Lab (e.g. arm angles, velocities, accelerations) or any of the data created by Kinarm supplied peripherals. Those types of data are automatically saved with your recorded data. Be sure to familiarize yourself with the data that is recorded automatically by your Kinarm Lab before saving data as custom analog inputs. Dexterit-E Explorer can be used to review a collected exam and the kinematics that are recorded

10.5 Loads on the Kinarm Robot

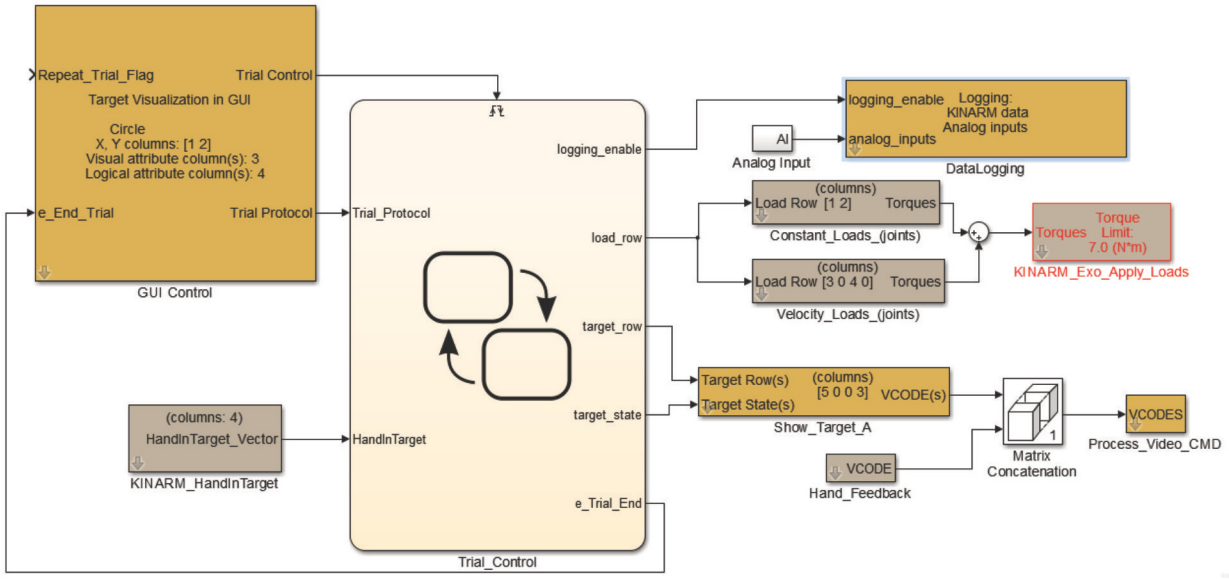
This section will explain the Simulink blocks necessary to apply loads to a Kinarm robot. In this example, the sum of two different classes of loads are to be applied to a Kinarm Exoskeleton robot: constant loads and velocity dependent (e.g., viscous or curl field) loads.

The Simulink blocks used to create and apply loads in this example are available in the `KINARM Exo loads` library of the Task Development Kit. For Kinarm End-Point robots, analogous blocks are available in the `KINARM EP loads` library.

10.5.1 Simulink Code for Loads on the Kinarm Robot Example

Implementing a load on the Kinarm robot requires at least 2 blocks: a load block that creates a torque output (e.g. `Constant_Loads_(joints)`) and either the `KINARM_Exo_Apply_Loads` block or the `KINARM_EP_Apply_Loads` block, which apply the desired torques to the Kinarm robot. Multiple simultaneous loads can be applied as shown in the example below, by copying additional load blocks from the Task Development Kit library and summing the output torques prior to inputting the torque to the `KINARM_Exo|EP_Apply_Loads`block. The actual parameters to be used for each load are gathered from the Load Table, as referenced by the column indices chosen by the block (in this example, `Constant_Loads_(joints)` will reference columns [1 2]) and also by the `row` input to the block from the Stateflow chart (the Stateflow chart has a new output: `load_row`). For more details on load and column definitions, click Help for the "load" block of interest within Simulink.

Figure 10-10: Simulink Code for Example 7.5 - "Loads on the Kinarm Robot"



NOTE: There can be only one copy of each of the `KINARM_Exo_Apply_Loads` block or the `KINARM_EP_Apply_Loads` block in any Task Program.

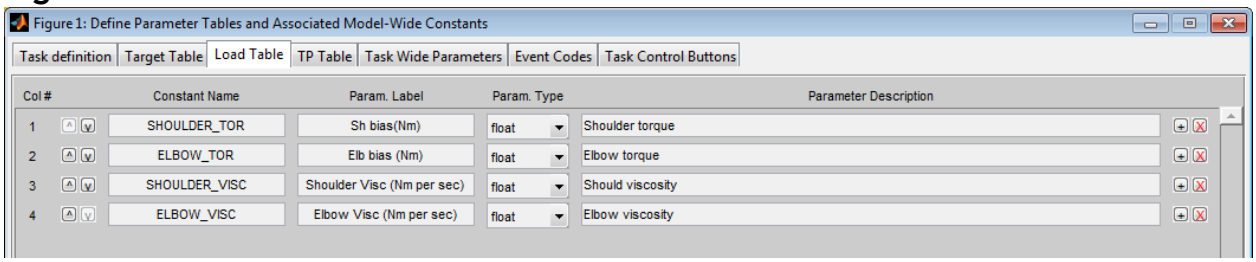
10.5.2 Stateflow Chart for Loads on the Kinarm Robot Example

The Stateflow Chart for this example is not shown here. The main feature is a new Stateflow output `load_row`, which must be set as desired from within Stateflow (e.g. `load_row = Trial_Protocol[LOAD_COL]`, which assumes that the load choice is defined in the `Trial_Protocol`).

10.5.3 Parameter Table Defn Block for Loads on the Kinarm Robot Example

Although the Simulink code completely defines how coefficients in the Load Table will get passed to the various Load blocks in the Task Program, in order for the Load Table to appear properly for editing in Dexterit-E's GUI, the `Parameter Table Defn` block needs to be edited. For each column in the Load Table that is to be used, a line needs to be included in the table definition GUI. Below are the four lines that would be included for this "Loads" example (only four columns from the Load Table are used by the Simulink code: columns 1 and 2 for the `Constant_Loads_(joint)` block and columns 3 and 4 for the `Velocity_Loads_(joint)` block).

Figure 10-11: Load Table Defnitions



- **Col#** – The column in the Load Table.
- **Constant Name** - A name that can be used within any Stateflow code, or in Simulink blocks as described in Section 3.12.
- **Param Label** - The label shown to the operator in the Dexterit-E GUI when editing a protocol's Load Table.
- **Param Type** – The type that the number should be recorded as in the Dexterit-E protocol editing GUI.
- **Parameter Description** – The description of the of the column that will be shown in the Dexterit-E GUI.

NOTE: Loads are applied to the Kinarm robot, not to the subject, whether the loads are hand-based or joint-based. Under static conditions the load applied to the Kinarm robot will equal the load applied to the subject, but under dynamic conditions these will not be equal because of Newton's laws and the equations of motion.

NOTE: The constant names defined in the `Parameter Table Defn` block can be used to specify the columns in the various load blocks.

10.6 SLRT Scopes

This example shows the use of an SLRT Scope in a Task Program. SLRT Scopes allow the operator to view signals during Task Program execution on the Robot Computer. This feature can be useful when debugging a Task Program. In a typical setup, where the Robot Computer and Dexterit-E computer share a keyboard, video and mouse, the KVM switch will need to be used to switch the monitor to view the Robot Computer (e.g. double tap <Scroll Lock> to switch which computer is viewed on the monitor).

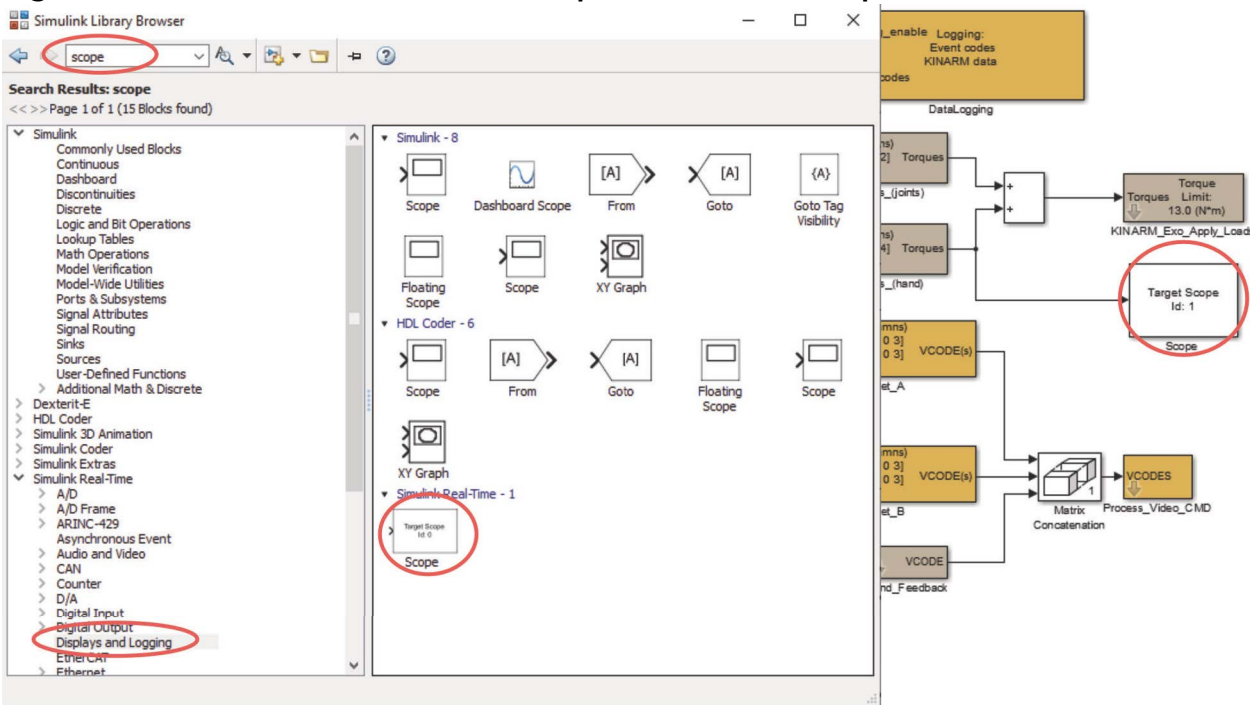
NOTE: The SLRT Scope block should not be confused with the standard Simulink Scope block.

10.6.1 Simulink Code for SLRT Scopes Example

To use an SLRT Scope, the block can be dragged from the Simulink Library Browser; the block is found at **Simulink Real-Time** -> **Displays and Logging** -> `Scope` . It is also possible to just search for 'scope' in the search box.

- Once it has been dragged into the Task Program, a signal can be connected to the SLRT Scope.
- To view multiple signals on the same scope, the signals must be muxed together first. Up to 10 signals can be seen on a single scope.
- To view different signals on different scopes, drag multiple copies of the SLRT Scope block into the Task Program. Up to 9 SLRT Scopes can be in a single model.
- Double-clicking on the SLRT Scope will bring up a dialog (not shown) with numerous options, such as whether to show numerical or graphical data, number of samples to show, triggering options etc.

Figure 10-12: Simulink Code for Example 7.6 - "SLRT Scopes"



10.7 Multiple Targets and Multiple Target States

This example shows how multiple targets can be displayed simultaneously and how multiple target states can be used. The purpose of having multiple "on" states for a target is to allow a single target to change from one colour and/or size to another without having to completely redefine the target. For example, if you wish to have a target change from green to red, then a target can be defined with two states such that the two states share the same location, size and target type, and the only difference between them is the colour.

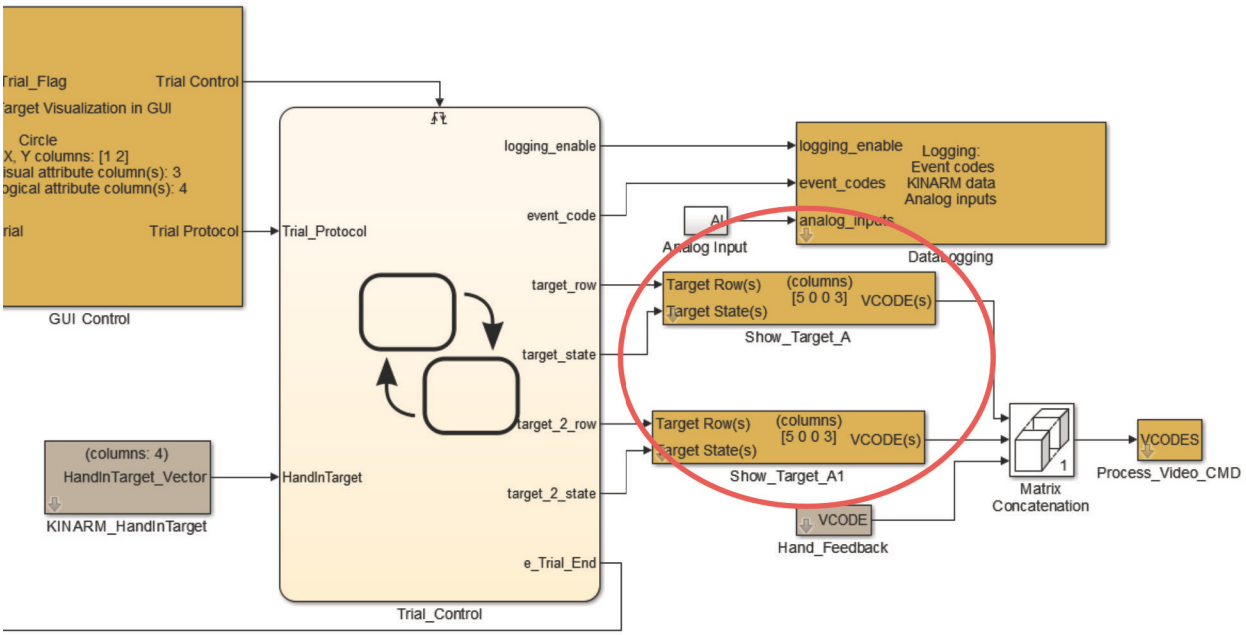
10.7.1 Simulink Code for Multiple Targets and Multiple Target States Example

Multiple states for a target are defined by double-clicking the `Show_Target` block within Simulink, and choosing 2 or more states. For each desired state, the columns in the `Target_Table` that reference size and colour need to be chosen. Having the target switch back and forth between these states then requires the `Target_State` input to the `Show_Target`block to be the desired state. For more information on defining target states, click Help in the `Show_Target`block from within Simulink.

Showing multiple targets simultaneously requires two things: the `Matrix Concatenation` block (which concatenates the `VCODE` outputs) and a separate `Show_Target` block for each target. In this particular example, 2 targets are shown.

You can either set the transparency (or alpha) for a target within a Simulink `Show_Target`block, or have it passed into the `Show_Target` block as a parameter. Within the `Show_Target` block you can also specify if a target should show up on the operator display (i.e. in Dexterit-E), the subject display, or both.

Figure 10-13: Simulink Code for Example 7.7 - "Multiple Targets and Multiple Target States"



10.7.2 Stateflow Chart for Multiple Targets and Multiple Target States Example

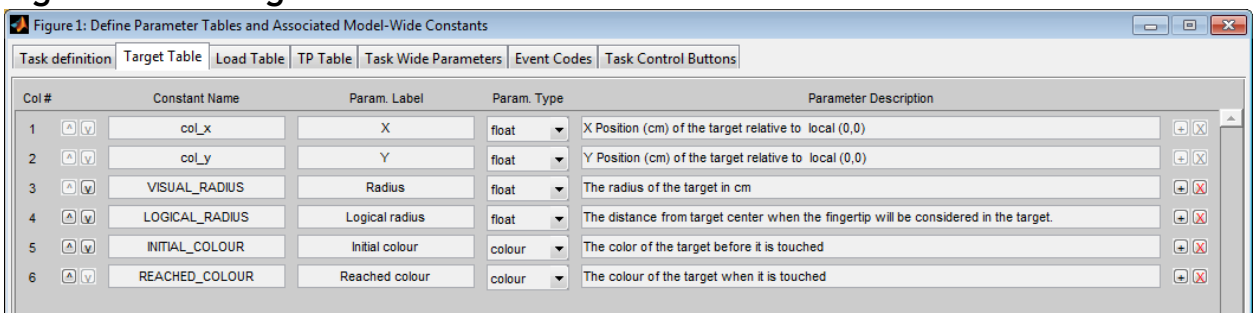
The Stateflow Chart for this example is not shown here. The main features are two new outputs `target_2_row` and `target_2_state` which must be set as desired from within Stateflow in a similar manner to target_row and target_state. Also note that `target_state` and `target_state_2`now have three valid values: 0 (target off), 1 (state 1) and 2 (state 2).

NOTE: In this example there are 2 `Show_Target` blocks. However, the `Show_Target` block can take a vector input such that it will display more than one target at a time, which allows you to accomplish what is done here with a single `Show_Target` block.

10.7.3 Parameter Table Defn Block for Multiple Targets and Multiple Target States Example

In this example, another column has been defined in the Target Table (*Figure: 10-14*). State 2 of the `Show_Target` blocks reference column 6 of the Target Table for the colour of the targets (see Simulink code above) and so a new line needs to be added to the `Parameter Table Defn`.

Figure 10-14: Target Table Definition



NOTE: The constant names defined in the `Parameter Table Defn` block can be used to specify the columns in the various target blocks.

10.8 Background Targets

Background Targets (or permanent targets) are an option that is primarily maintained for legacy purposes, because earlier releases of Dexterit-E were much more limited in the number of visual stimuli that could be presented dynamically every frame. However, there may still be circumstances in which the limits of the system are reached (e.g. there is a maximum number of VCodes that can be sent each frame, as specified in Section *Section: 14.7 VCodes - Programming Visual Stimuli*). If you have reached the limit and need to display more targets, then you can consider using background targets. These are targets which are defined just like any other target, but they are pushed through a `Set_Target_in_Background` block once, and do not need to be re-sent each frame.

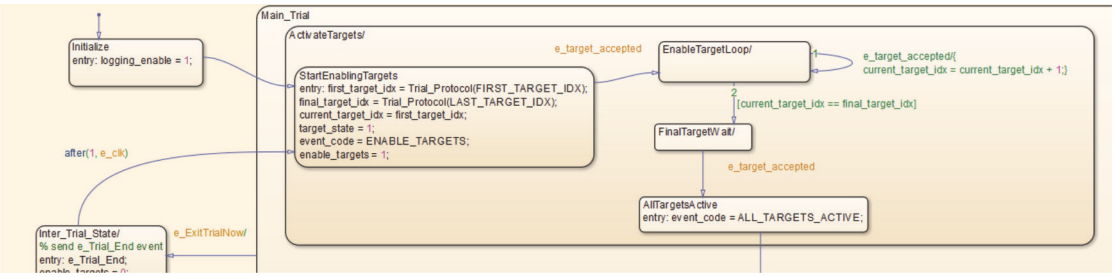
10.8.1 Simulink Code for Background Targets Example

This example shows how to use the `Set_Target_in_Background` block. It requires the VCode that is created in a normal `Show_Target` block, an ID to specify which background target to deal with, and a state to turn it on or off. In addition, the Task Program needs to monitor the status of the state changes to ensure that they have been implemented.

This monitoring of state changes is handled by the `Background VCODE Received` signal from the `Set_Target_in_Background` block, which in this example is fed back into the Stateflow diagram. This signal is a pulse which indicates that the state update has been received and you can continue. This feedback signal goes to 1 for a single clock cycle when a background target command has been received and processed by the Dexterit-E Computer. Waiting for this feedback signal is required, otherwise any changes you make to the target are not likely to take effect. Generally the feedback signal should indicate success in less than 1 V-sync (i.e. < 16.7 ms for a 60 Hz display).

Here is an example of the Stateflow code that you would use to turn on a set of targets:

Figure 10-15: Simulink Code for Example 7.8



It is convenient here to use input signals to drive the enabling of targets. When using signals in this manner it is important to have the signal only trigger on a rising edge.

10.9 Targets with Text

You can display targets with text. To do this you need to define 3 columns for the Target Table in the `Parameter Table Defn` block, with specific options set for the **Param. Type**. One column needs to have a **Param. Type** of *label* (for the text to be displayed), one of *colour* (for the text colour), and the third of *float* (for the size of the text in cm). These data need to be passed to a `Show_Target_With_Label` Simulink block within your model. Within Dexterit-E you will be able to define the text. The text will always be shown centred over the target you have specified and at the size you have specified. The text can only be a maximum of 50 characters long.

Showing dynamic text (eg. a score) is a more complex task that generally requires coding in a Simulink embedded MATLAB block. The Kinarm support web site provides some examples of tasks showing dynamic text. As well, you can review for examples of how to manipulate VCodes in embedded MATLAB blocks.

10.10 Images as Targets

It is possible to show a JPG, PNG, or BMP as a target. This is mainly handled through Dexterit-E. When you define your Target Table in the `Parameter Table Defn` block, any column with a **Param. Type** of *colour* can be used within the protocol editor in Dexterit-E to select images or colours to display. Only images which are stored in the task's directory will be available. The image will be displayed mapped onto the shape you have specified in the Target Table. If you need to manipulate VCodes manually in the Task Program (e.g. via a MATLAB function) then be aware that negative values assigned to a fill colour are interpreted as image indexes. Images are indexed in alphabetical order within the task directory. For complete details on VCodes see *Section: 14.7 VCodes - Programming Visual Stimuli*.

NOTE: Dexterit-E will only load the first 400 MB of images, further images will not be loaded. Images larger than 50 MB will not be loaded. These sizes are calculated as video memory size, not file size. Video memory size is: image width in pixels * image height in pixels * 4 bytes per pixel.

NOTE: Interlaced PNG's are not supported.

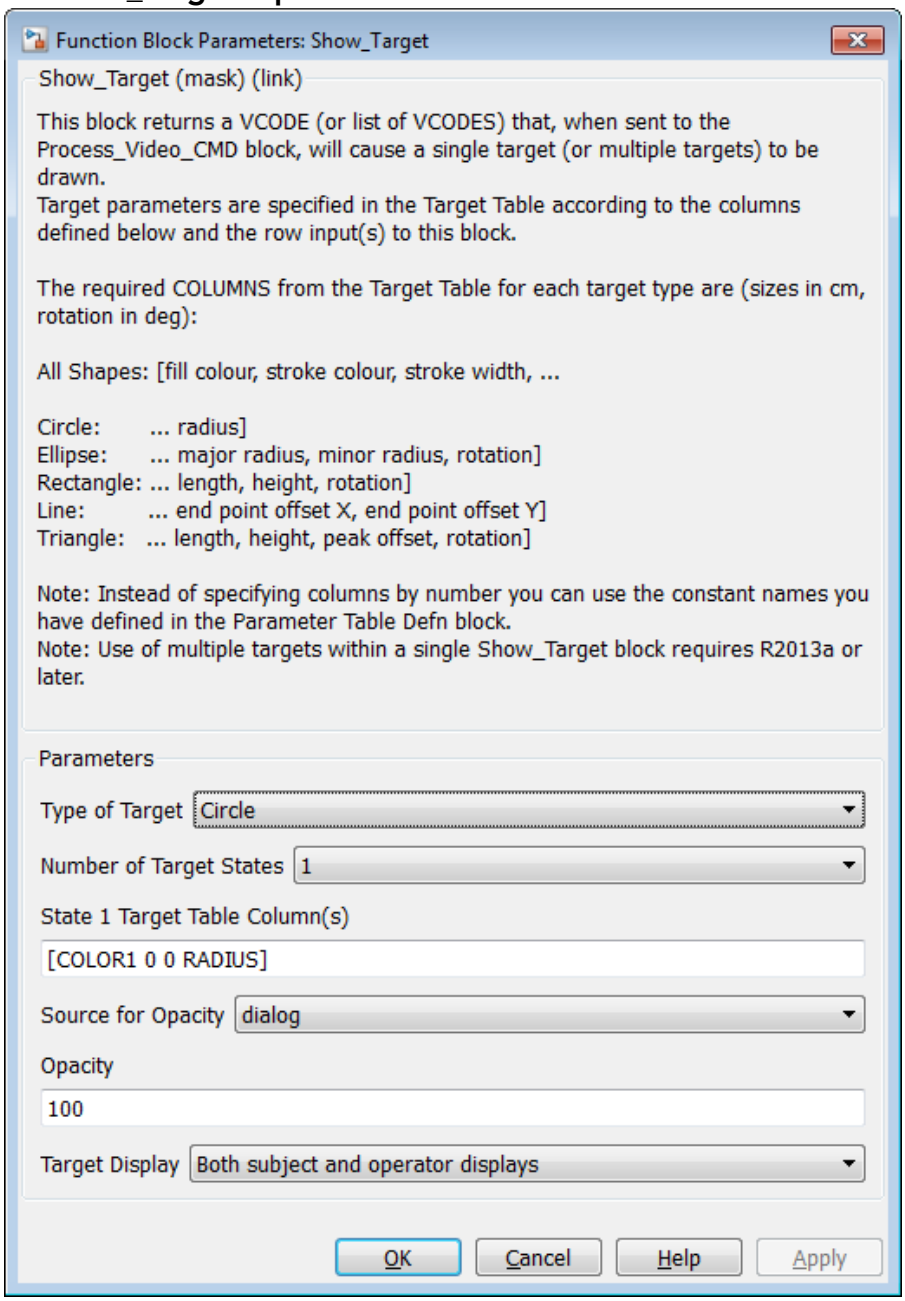
10.11 Selective Target Display Options

It is normal that most targets are shown both on the subject display and within Dexterit-E. However, it is possible to display targets only to the subject, or only to the operator in Dexterit-E. These are options which are set within the block mask for a `Show_Target` or `Show_Target_With_Label` block.

It is also possible to apply a transparency setting to targets within the `Show_Target` or `Show_Target_With_Label` block. This allows one target to be seen underneath another.

If you would like a target to have just a perimeter drawn then you can use the special "No Fill" colour "16777216" in the Target Table. Within the Dexterit-E GUI if you are editing the Target Table for a Task Protocol the colour editing dialog has a **No fill** button that will handle creating this special colour for you. Clicking **No fill** has the effect of not applying a fill to the object (i.e. the object is transparent). If the stroke colour is specified as the **No fill** colour then the perimeter of the target is not drawn (i.e. it is transparent).

Figure 10-16: Show_Target Options



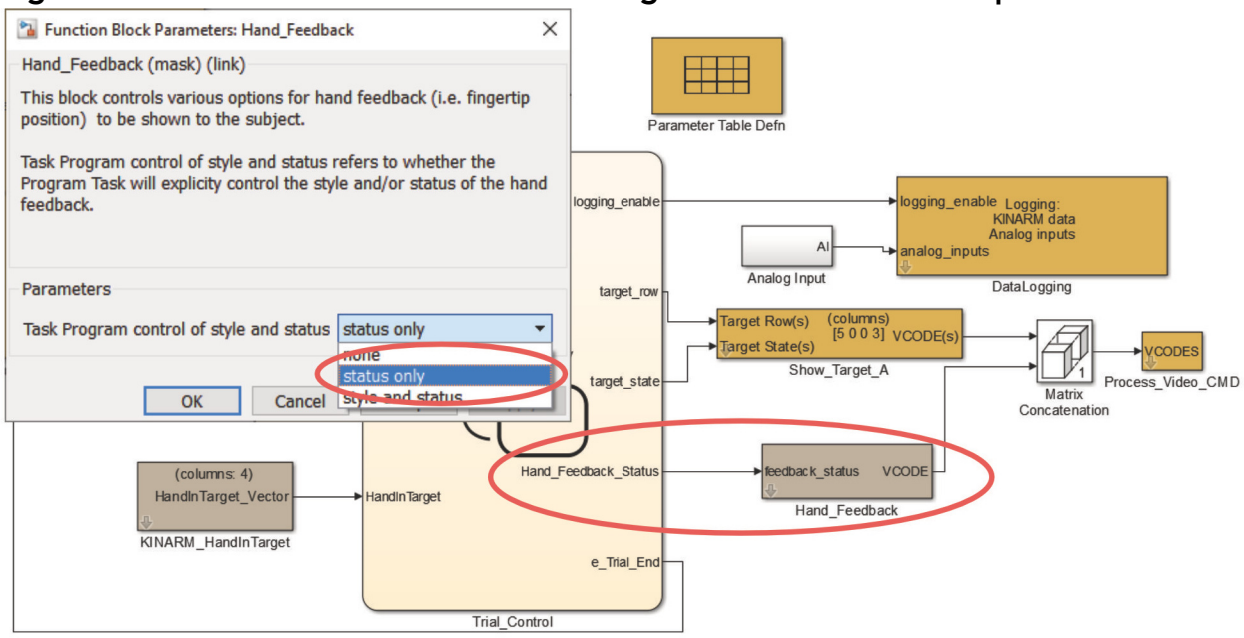
10.12 Controlling Hand Feedback

This example demonstrates how visual hand feedback can be controlled during a task (e.g. on for some trials and off for others, or changing on/off part-way through a trial). In order to understand what will actually occur during a task, it is important to realize that hand feedback control in Dexterit-E is multi-tiered. Dexterit-E allows an operator to choose the control of hand feedback either as on, off or controlled by the Task Program. The settings chosen within Dexterit-E will override those chosen within the Task Program, so the Task Program can only control hand feedback if the operator has chosen **Task Program Control** for hand feedback within Dexterit-E (see the Dexterit-E User Guide for more information).

10.12.1 Simulink Code for Controlling Hand Feedback Example

From within the **Hand_Feedback** block there are various options for choosing hand feedback control. In this example, Task Program control of **status only** has been chosen, this means that the status, for example the on/off state, of hand feedback can be controlled by the Task Program through the `feedback_status` input. For more information on the hand feedback control options, click **Help** for the `Hand_Feedback` block within Simulink.

Figure 10-17: Simulink Code for Controlling Hand Feedback Example



10.12.2 Stateflow Chart for Controlling Hand Feedback Example

The Stateflow Chart for this example is not shown here. The main feature is a new output `Hand_Feedback_Status` which must be set as desired from within Stateflow. (e.g. `Hand_Feedback_Status=1`).

10.13 Error Trials: Repeating and/or Reporting

This example demonstrates how a trial can be forced to be repeated. In this context, an error trial is a trial in which the subject has done something incorrectly (e.g. did not reach to a target). In this example, we set up the task to both repeat and record the error, however, only one or the other need be done. In other words they are independent from each other, so if both types of functionality are desired, then both need to be implemented.

In order to potentially repeat the trial (which will occur at the end of the current block), the `Repeat_Trial_Flag` input of the `GUI Control` block is used. This input is only available if Show repeat trial flag has been checked in the `GUI Control` dialog. Its purpose being to allow the option for a trial to be repeated if desired. However, in order to understand what will actually occur during a task, it is important to realize that Repeat Error Trials control in Dexterit-E is multi-tiered. Dexterit-E allows an operator to choose whether or not the `Repeat_Trial_Flag` is ignored using the Repeat Error Trials checkbox from within the Task Protocol editor. The table below shows the combinations under which a trial is or is not repeated.

Table 10-2: Conditions for Repeating a Trial

		Repeat_Trial_Flag value sent to GUI Control block	
		Repeat_Trial_Flag=0	Repeat_Trial_Flag=1
Repeat Error Trials checkbox in Dexterit-E protocol editor	checked	Not repeated	Repeated
	unchecked	Not repeated	Not repeated

In order to report a trial as being an error trial to the Dexterit-E User Interface, a Task Event Code has to be created to report the error and that Task Event Code needs to be defined as error-reporting. Task Event Codes can be set as error-reporting in the `Parameter Table Dfn` block by setting the Options field to error for the Task Event Code. Multiple error-reporting Task Event Codes can be defined for a single task. If any error-reporting Task Event Code occurs during a trial, Dexterit-E records that trial as an error-trial and the percent correct statistics in the Dexterit-E window are updated appropriately.

Figure 10-18: Parameter Table Defn Block Error - Reporting Task Event Code

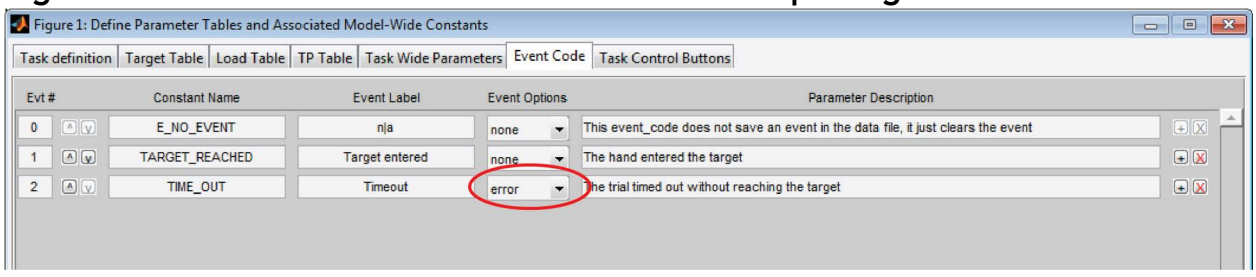
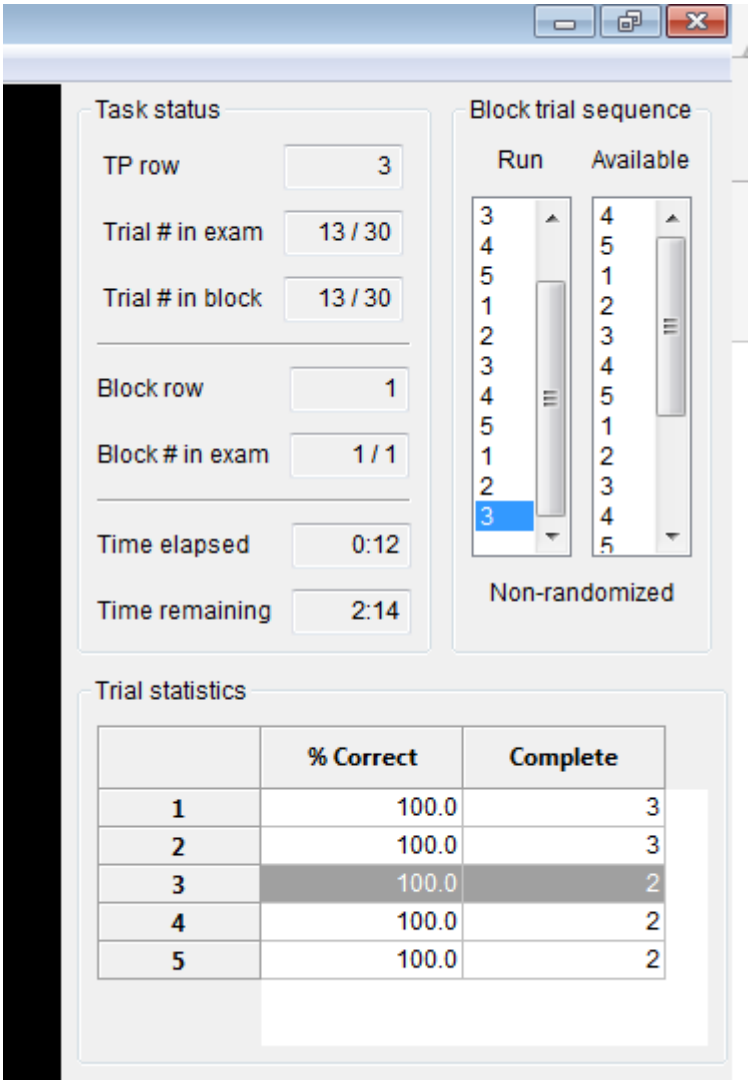


Figure 10-19: Dexterit-E Trial Statistics Window

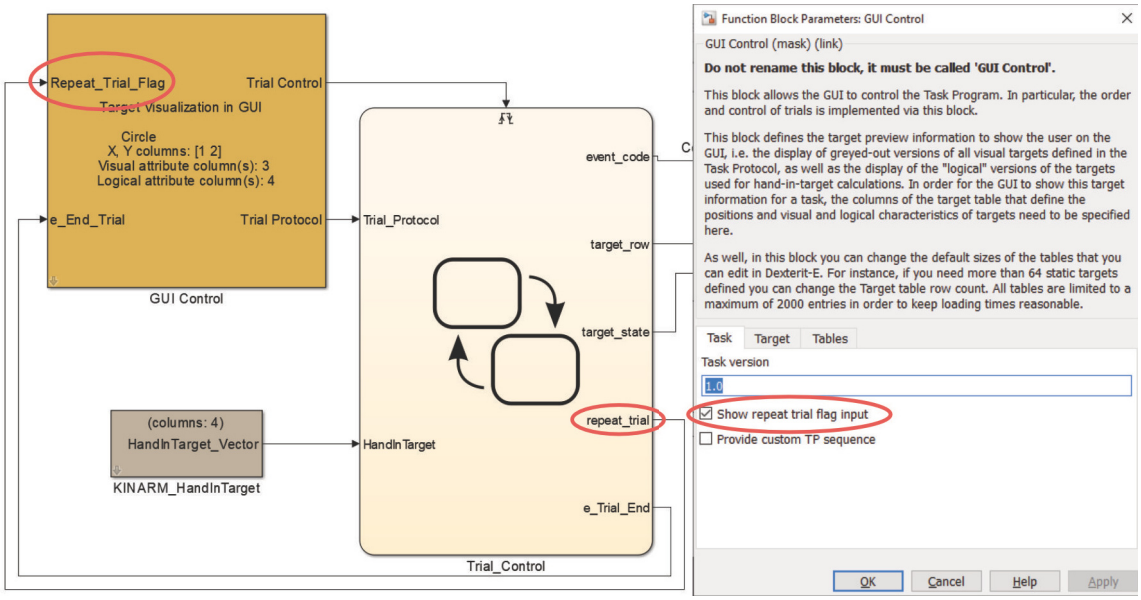


NOTE: If your task does not have any Task Event Codes defined with the error option, then the Trial Statistics window will be hidden.

10.13.1 Simulink Code for Error Trials: Repeating and/or Reporting Example

In this example, a connection is made from the Stateflow chart to the `Repeat_Trial_Flag` input of the `GUI Control` block, after checking **Show repeat trial flag input**.

Figure 10-20: Simulink Code for Error Trials: Repeating and/or Reporting Example

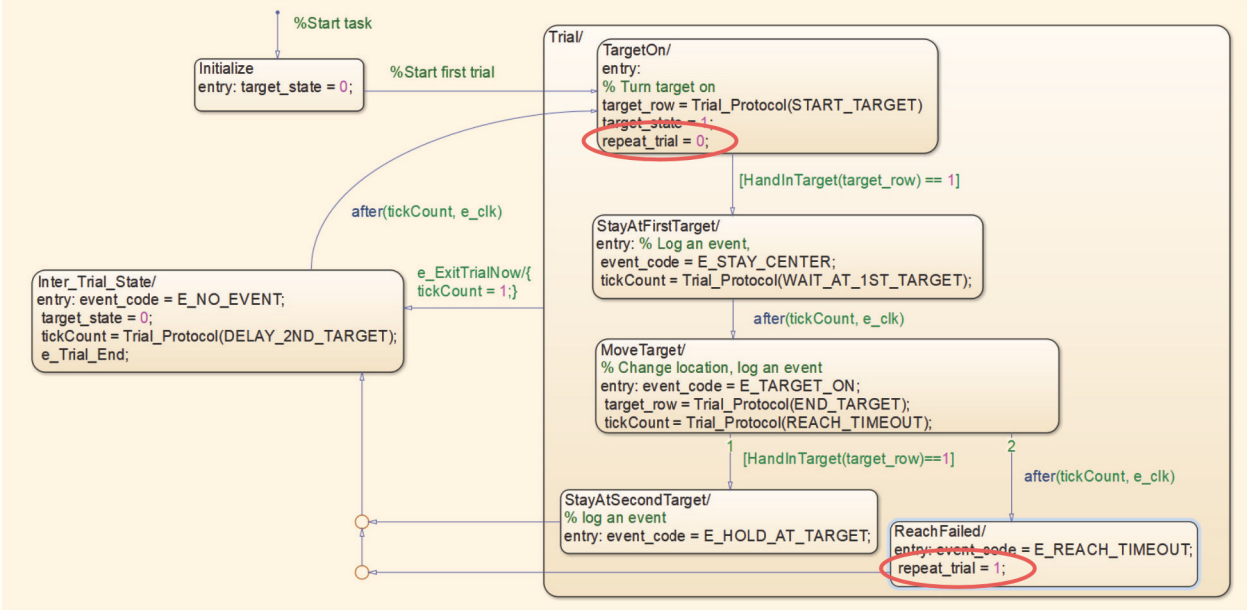


10.13.2 Stateflow Chart for Error Trials Repeating or Reporting Example

In order to repeat an error trial, the new Stateflow output `repeat_trial` must be set as desired from within Stateflow (e.g. if something does not occur as desired, then set `repeat_trial=1`). In the example shown below, `repeat_trial` is set =0 at the start each trial. If the subject does not reach to the second target before a given time has expired (i.e. from `tickCount = Trial_Protocol(REACH_TIMEOUT)`), then `repeat_trial=1` is set.

If the subject does not reach to the second target before a given time has expired then `event_code=E_REACH_TIMEOUT` is also set. This Task Event Code will be stored as part of the trial's data file and so can be used by subsequent data analysis to indicate that this trial was an error trial. Stateflow Code for *Section: 10.13 Error Trials: Repeating and/or Reporting*.

Figure 10-21: Stateflow Chart for Error Trials: Repeating and/or Reporting Example



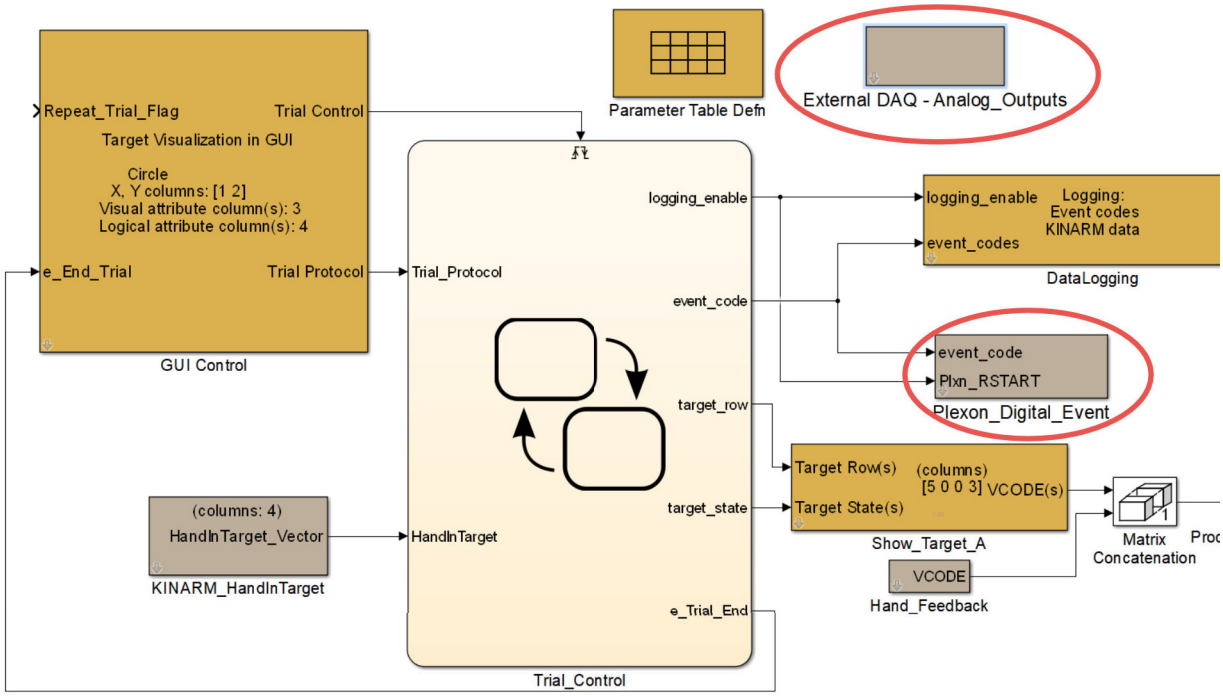
10.14 External Data Acquisition System

This example shows what changes are necessary to have a Task Program function with an external data acquisition system (DAQ). The purpose of such a setup is to have a central data acquisition system rather than multiple systems which would require post-experiment synchronization of data files (e.g. if using a neural recording system such as Plexon). This type of setup requires different hardware than the standard Dexterit-E requirements. Refer to the Dexterit-E User's Guide for more information.

10.14.1 Simulink Code for External Data Acquisition System Example

In this example, two new Simulink blocks are required to send information over to the external data acquisition system. Analog information relating to Kinarm robot kinematics is sent using the `External DAQ-Analog_Outputs` block (i.e. joint position, velocity and acceleration) while Task Event Code related information is sent using the `Plexon_Digital_Event` block (this is a Plexon-specific example). For more information on interfacing to an external DAQ system with these Simulink blocks, please click Help for those blocks, available from within Simulink.

Figure 10-22: Simulink Code for External Data Acquisition System Example



NOTE: The reason for sending velocity and acceleration data to an external system in addition to position is that the conversion from digital to analog (on the Robot Computer) and then back from analog to digital (in the external DAQ system) introduces quantization error into the position signal. This quantization error is very small (<0.05% for standard 12-bit systems) but it becomes magnified upon subsequent differentiation. Refer to the Dexterit-E User's Guide for more information.

10.15 Bilateral Kinarm Lab Feedback and Loads

This example shows some basics of bilateral Kinarm task control, including hand feedback for both arms, and loading conditions on both arms.

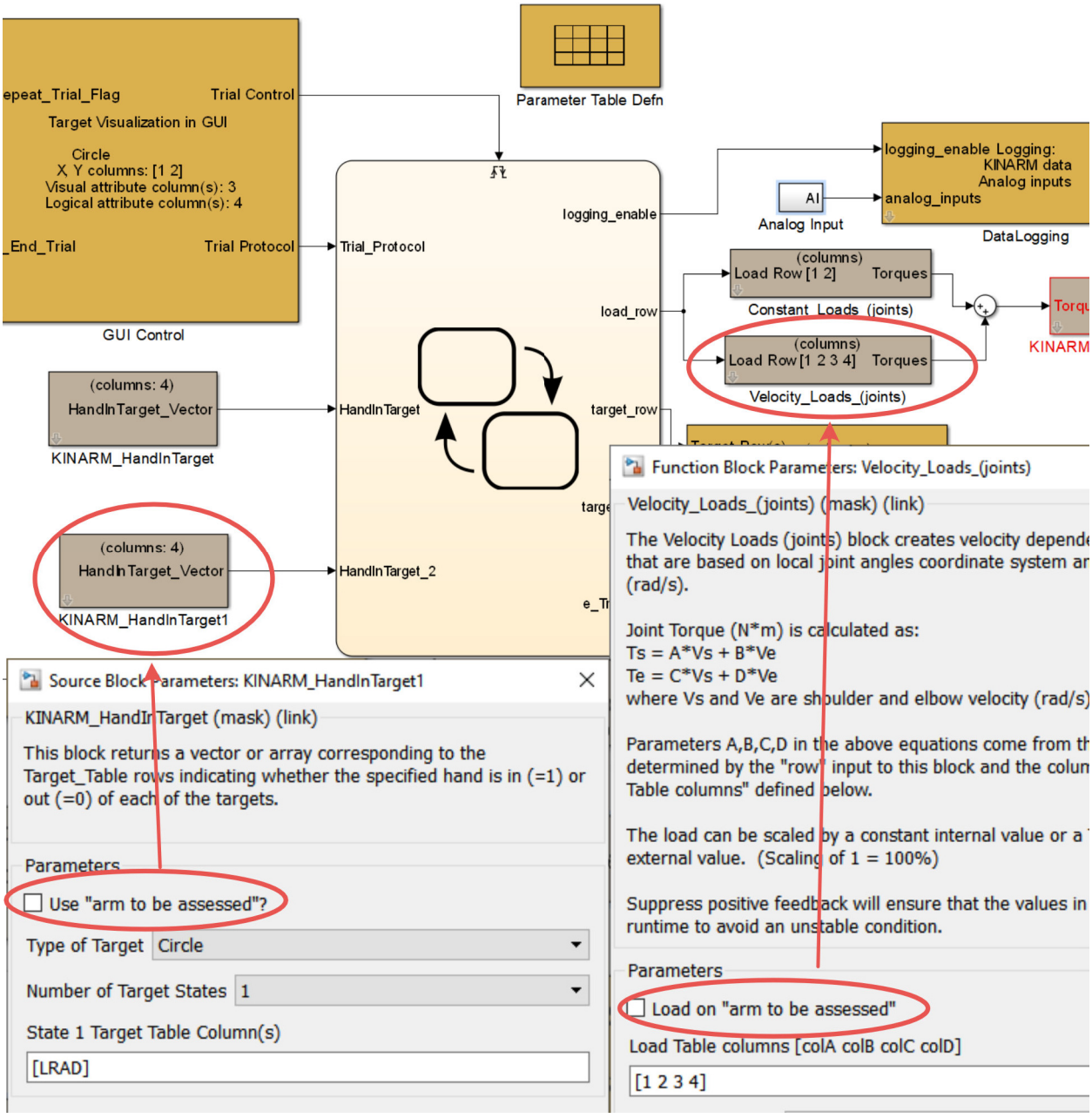
10.15.1 Simulink Code for Bilateral Kinarm Lab Feedback and Loads Example

In this example, there are two copies of the `KINARM_HandInTarget` block (where there was only one in *Section: 10.5 Loads on the Kinarm Robot*). Furthermore, as explained in more detail below, the parameters for the blocks circled in the image below have also changed.

The only difference between the two circled blocks and their uncircled counterparts is the unchecked box for **Use arm to be assessed**. When a Task Program and Task Protocol are selected in Dexterit-E, the operator has the option of selecting the **Arm to be assessed** as either the right arm or the left arm. The meaning of **Arm to be assessed** is Task Program specific, as defined by blocks such as these. The contralateral arm is thus

the other arm. Task Programs in Simulink are therefore not defined in terms of right-hand or left-handed, but rather in terms of an "arm to be assessed" and a "contralateral arm". If the operator selects a Task Protocol in Dexterit-E and checks Right for the Arm to be assessed, that means that the `KINARM_HandInTarget` is going to output what the right arm is doing, while `KINARM_HandInTarget_2` is going to output what the left arm is doing (i.e. the contralateral arm). If the operator chooses a Task Protocol for this same Task Program, but then checks Left for the **Arm to be assessed**, then `KINARM_HandInTarget` is going to output what the left arm is doing, while `KINARM_HandInTarget_2` is going to output what the right arm is doing.

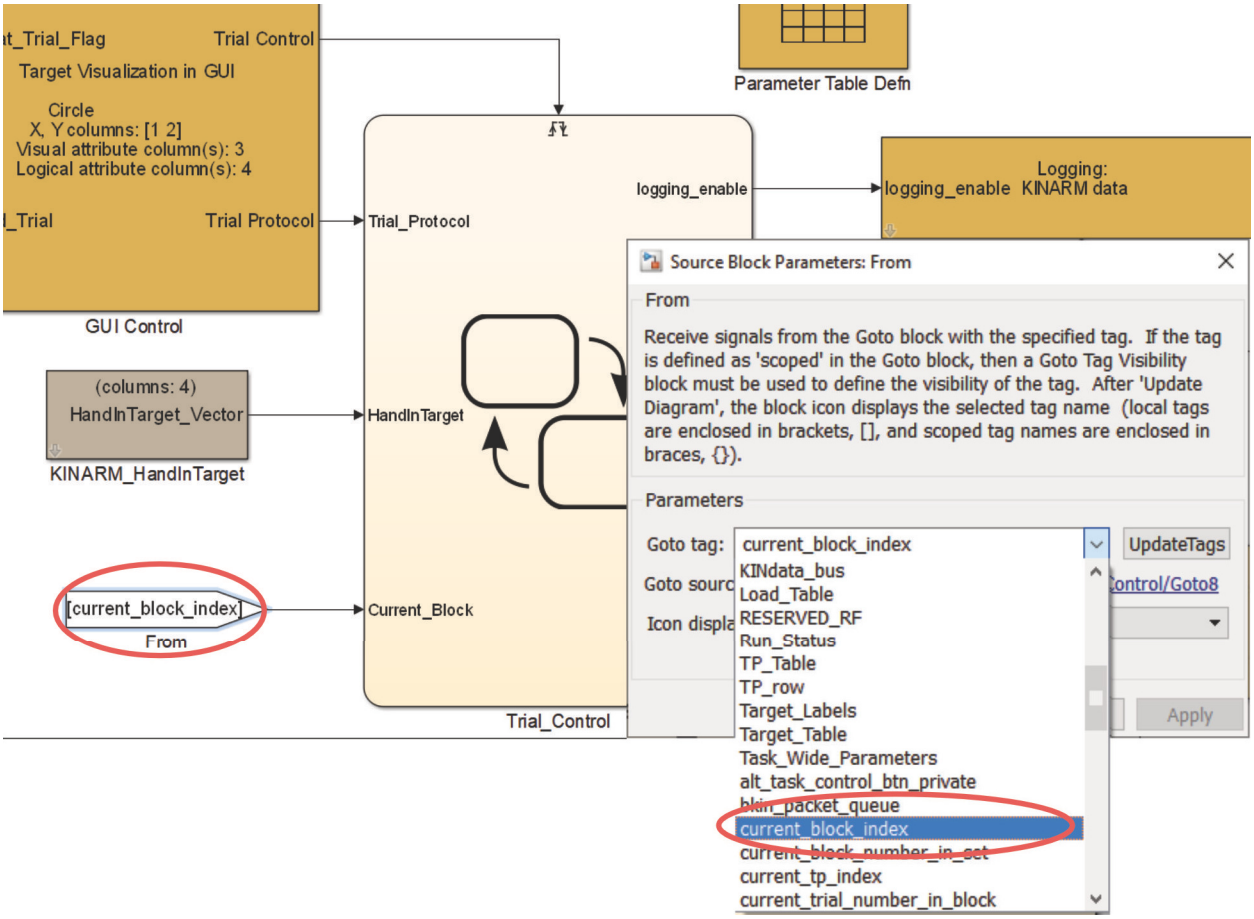
Figure 10-23: Simulink Code for Bilateral Kinarm Lab Feedback and Loads Example



10.16 Accessing KINdata_bus, Current Block Index, and Other Task Control Variables

It is possible to access various task control variables used by the Simulink blocks that make up the Dexterit-E library through the use of the Simulink's `From` block (see Simulink\Signal Routing library). The `From` block creates an invisible connection to a `Goto` block elsewhere in the Simulink model (multiple `From` blocks can access a single `Goto` block). Accessing various task control variables can be desirable, for example, if something needs to be done only on the first trial (in which `current_trial_number_in_set` would be a useful variable to access) or if a custom block is being created that requires access to various Kinarm data (in which case `KINdata_bus` would be the relevant variable to access). Paste the `From` block into a model, double-click it, click **Update Tags** and then select the **Goto Tag** options to see a list of the available parameters that can be read from. These blocks allow the end-user to access things such as the various parameters tables (e.g. `Target_Table`) as well as status of the overall task (e.g. `current_block_index` , `current_tp_index`). For more information regarding which `Goto` tags are available, please see *Section: 14.9 Available 'Tags' (From and Goto Blocks)*.

Figure 10-24: Simulink Chart for Accessing KINdata_bus, Current Block Index, and Other Task Control Variables



NOTE: To find out more about what each of the possible variables are, choose the relevant Goto tag, then click Goto Source in the `From` block dialog to open the Simulink block in which the `Goto` block resides. Also, you can review the available tags in *Section: 14.9 Available 'Tags' (From and Goto Blocks)*.

10.17 Multiple Conditions for Stateflow Transitions (and `e_clk` event)

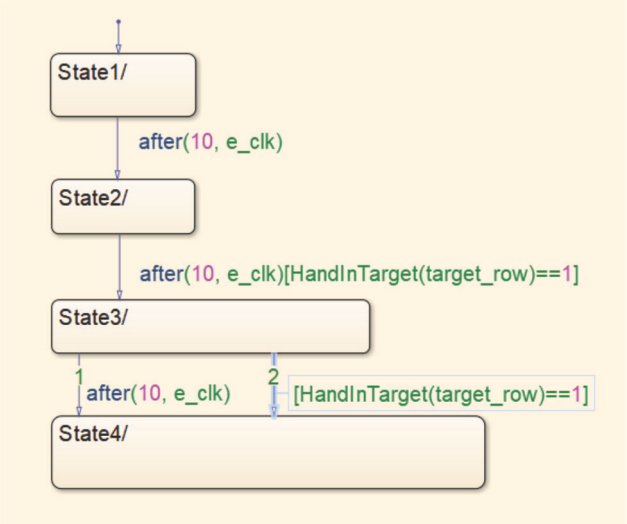
This example demonstrates a few of the more common transition conditions that end-users are likely to use and how to combine multiple conditions appropriately. For more information on transitions, refer to the MathWorks Stateflow documentation.

10.17.1 Stateflow Chart for Multiple Conditions for Stateflow Transitions and `e_clk` Event Example

In the example shown here, we have assumed that the `e_clk` Stateflow event is defined in Stateflow's Model Explorer as per the sample Task Programs included with Dexterit-E. Namely, `e_clk` is the first input Stateflow event and is triggered only on the rising edge. In this manner, `e_clk` events in a Task Program will occur once every 1 ms.

For the Stateflow chart below, the transition from `State1` to `State2` occurs after 10 e_clk events have occurred (i.e 10 ms). The transition from `State2` to `State3` occurs only if the `HandInTarget` condition is true AND at least 10 `e_clk` events have occurred (i.e. both conditions must be true). The transition from `State3` to `State4` occurs if the `HandInTarget` condition is true OR 10 `e_clk` events have occurred (i.e. whichever condition occurs first).

Figure 10-25: Stateflow Chart for Multiple Conditions for Stateflow Transitions (and e_clk Event) Example

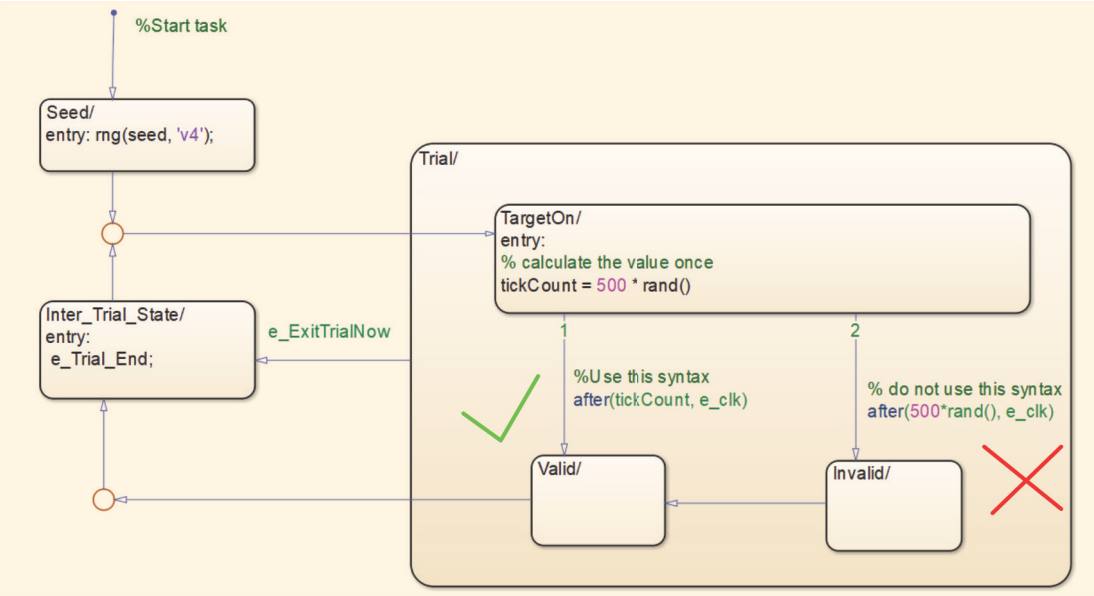


10.18 Random Numbers in Stateflow

The `rand()` function in Stateflow is typically used to calculate random numbers in a state and not in the evaluation of a transition. In the figure below you can see the correct and incorrect usages of `rand()`. in the example, the goal is to wait for a evenly distributed random period of time. Using `rand()` during a transition will cause `rand()` to be called each time that the transition is evaluated. In other words, the `after()` function in transition 2 compares the number of `e_clk` events to a different random number each time the transition is evaluated, whereas the `after()` function in transition 1 compares the number of `e_clk` events to the same random number each time that the transition is evaluated (with that same random number only changing when the `TargetOn` state is

entered). This difference means that as time goes on, the cumulative probability that transition 2 will become true will increase relative to transition 1. So transition 2 will usually be true before transition 1 and you will not get an evenly distributed wait period from 0-500 ms. Instead you will get a wait period from 0-500 ms that is highly skewed to lower values.

Figure 10-26: Using rand() in Stateflow



The function `rand()` returns a double between 0 and 1. In the initial state of the Stateflow chart the random number generator is seeded using the seed provided by Dexterit-E (the seed is obtained from the `From` tag "seed"). Dexterit-E provides a new random seed each time a task is run. When a Task Protocol is created in Dexterit-E, the default behaviour is for this seed to come from the current time, however, there is an option in the Task Protocol to use a constant seed such that the same sequence of random numbers is used each time a task is run.

Any of the standard MATLAB random number generating functions can be used in Stateflow.

NOTE: The above syntax is for Stateflow charts using MATLAB as the action language. If your Stateflow chart uses C as the action language then you will need to refer to earlier versions of this guide for instructions on using random numbers.

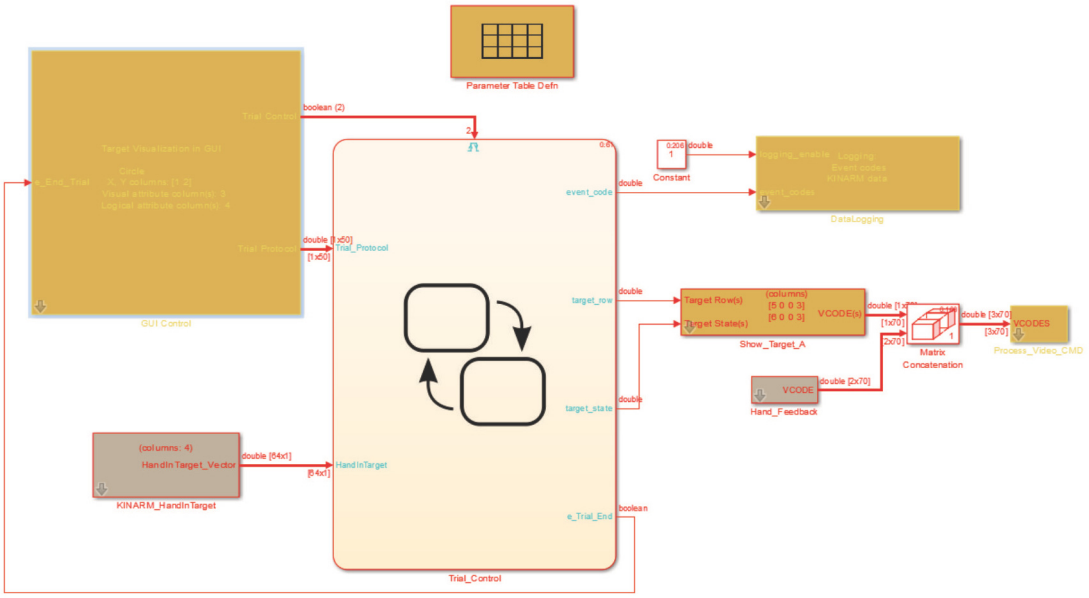
10.19 Parallel State Execution (Independent State Machines)

This example shows how to set up independent state machines that operate in parallel. In this example, control over the target (i.e. target colour) is controlled independently from the main flow of states in the task.

10.19.1 Simulink Code for Parallel State Execution (Independent State Machines) Example

The Simulink code for this example is based on the code in *Section: 6.7 Centre-out Reaching Task*. The one difference is the addition of a second target state in the `Show_Target`block. Please see *Section: 10.7 Multiple Targets and Multiple Target States* for more details on creating a second target state.

Figure 10-27: Simulink Code for Parallel State Example

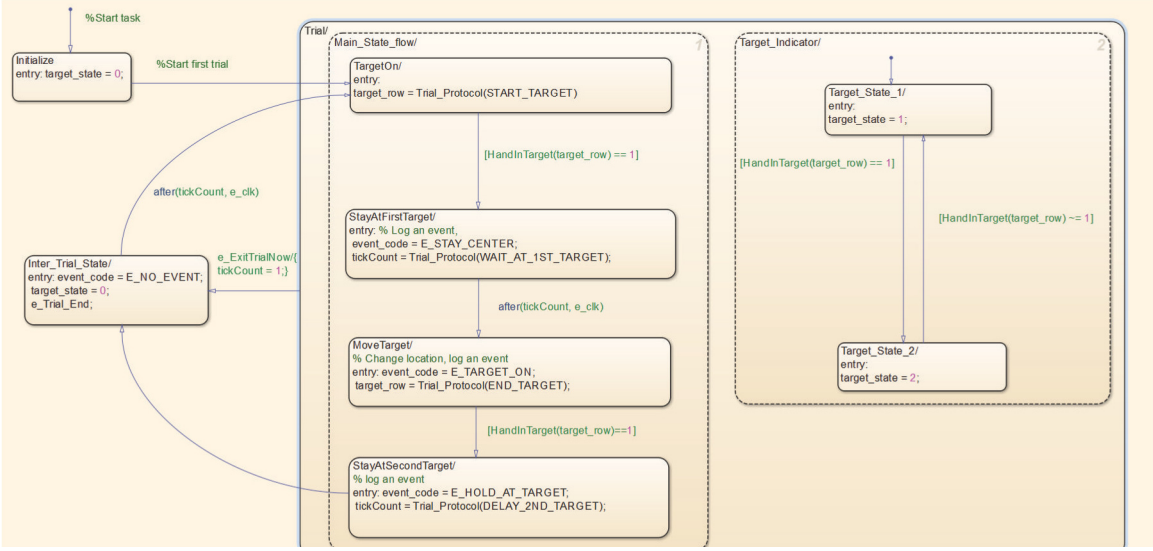


10.19.2 Stateflow Chart for Parallel State Execution (Independent State Machines) Example

The Stateflow Chart shown below is based upon *Section: 6.7 Centre-out Reaching Task*, however, several modifications were made to the chart:

- A new super-state, `Main_State_Flow`, was created encompassing most of the original states.
- A new state, `Target_Indicator`, was created along with two new sub-states, `Target_State_1` and `Target_State_2`.
- A new super-state, `Trial`, was created that encompasses the `Main_State_Flow` and `Target_Indicator` states.
- The decomposition of the super-state `Trial` was changed to Parallel (AND) by right-clicking inside of the super-state and selecting **Decomposition** -> **Parallel (AND)**. This step resulted in the `Main_State_Flow` and `Target_Indicator` states becoming dashed lines rather than solid lines to indicate that they will operate concurrently. The number in the upper right corner indicates which will execute first.

Figure 10-28: Stateflow Chart for Example 7.18 - "Random Numbers in Stateflow"



The above modifications mean that when a transition occurs into the `Trial` super-state, both the `Main_State_Flow` and `Target_Indicator` sub-states become active and both will operate and function concurrently. Likewise, when a transition occurs out of the `Trial` super-state, both the `Main_State_Flow` and `Target_Indicator` super-states become inactive and both cease to operate. For more complete information on using Parallel versus Exclusive states in Stateflow, please refer to the MathWorks Help documentation.

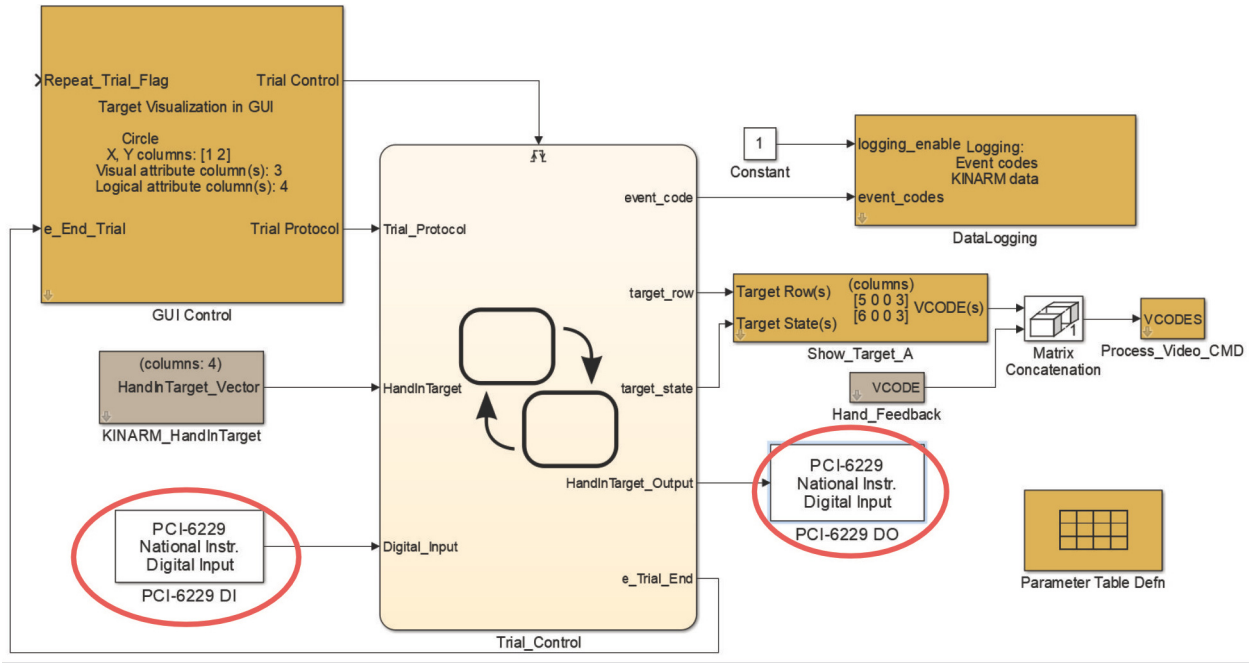
10.20 Digital Input/Output

This example shows how to set up digital inputs and outputs. In this example, a digital input is used to provide additional control over when a task transitions from one state to the next, and a digital output is used to indicate when a subject's hand is in or out of a target. The digital input could be from a push-button that the experimenter has and the digital output could go to a speaker to provide audio output.

10.20.1 Simulink Code for Digital I/O Example

The Simulink code for this example is based on the code in *Section: 6.7 Centre-out Reaching Task*. The differences are the addition a Digital Input block (`PCI-6229 DI`) and a Digital Output block (`PCI-6229 DO`), as well as the associated inputs/outputs in `Trial_Control`.

Figure 10-29: Simulink Code for Digital I/O Example

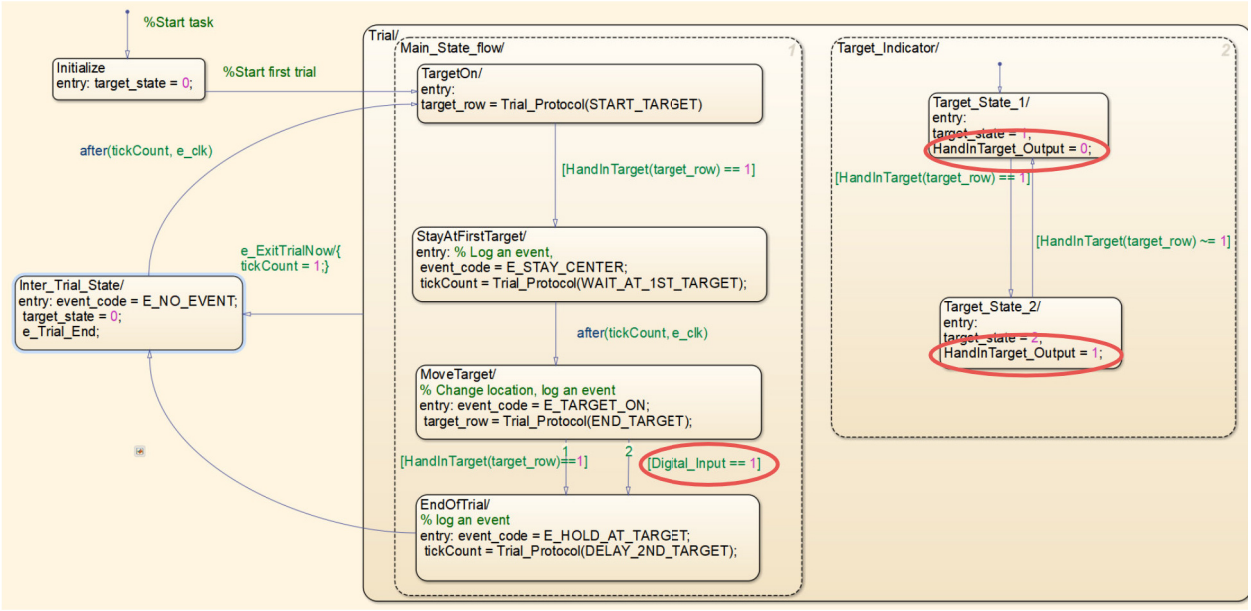


10.20.2 Stateflow Chart for Digital Input and Output Example

The Stateflow Chart shown below is based upon *Section: 10.19.2 Stateflow Chart for Parallel State Execution (Independent State Machines) Example*, with a few modifications:

- A new transition from `MoveTarget` to `EndOfTrial` was added, based on the condition `[Digital_Input==1]`. Thus if the push button connected to digital input 1 is pressed when the task is in the `MoveTarget` state, then `[Digital_Input==1]` will become true and the transition will occur.
- `HandInTarget_Output` is a newly defined output whose value depends on the `HandInTarget` value. If `[HandInTarget==1]`, then the Task Program will set `HandInTarget_Output=1`, which in this example is connected to a digital output that goes to a speaker, thus providing audio feedback on when a subject's hand is in the desired target.

Figure 10-30: Stateflow Chart for Example 7.20 - "Digital Input/Output"



10.20.3 Connecting to National Instruments I/O When Using a PCI-6229

Most Kinarm Labs come with a National Instruments PCI-6229 I/O card and a BNC 2090A to interface with it. The table below shows how the labels on the BNC 2090A relate to the I/O channels specified in Simulink for the PCI-6229.

Table 10-3: BNC 2090A to PCI-6229 interface

		Analog input		Analog Output		Digital I/O				PFI Digital I/O	
Simulink Channel Name		Ch 1-16	Ch 17-32	Ch 1-2	Ch 3,4	Ch 1-8	Ch 9-16	Ch 17	Ch 18-32	PFI 0	PFI 1-15
BNC 2090A Labels	Upper	AI 0-15		AO 0-1		PO 0-7					PFI 1-15
	Lower (optional)		AI 0-15		AO 0-1		PO 0-7		PFI 1-15		

10.21 Synchronization of Dexterit-E Data with External Clock

This example demonstrates how to provide a synchronization pulse that can be used to synchronize data collected by two independent data collection systems. Consider, for example, a system in which kinematic data were to be saved by Dexterit-E while simultaneously an external, independent data collection system were to save EEG data. In order to analyze relationships between the kinematic and EEG data, there would have to be a way to synchronize the data to be sure the events in one data set were related to events in the other.

The problem of synchronization is not necessarily an obvious one. It originates because no two clocks run at exactly the same rate. Two systems that both run nominally at 1 kHz, will always exhibit some amount of drift over time – i.e. one clock will always be slower or faster than the other. In some situations, the amount of drift will be negligible, and so synchronization between two data sets becomes trivial – if both data collection systems start on the same trigger, then the data will be synchronized. However, in most real-life situations, the amount of drift is large enough to be noticeable, and so it must be accounted for.

Many data collection systems avoid the problem of synchronizing multiple data sets by using a single master clock that drives all data collection systems (i.e. synchronization is forced by using only a single clock). This approach requires (a) that all but one of the data collections systems be capable of being driven by a master clock and (b) that the master clock pulse timing is precise and reliable. Many hardware systems do not meet these criteria, and so the problem of synchronizing two independent data sets remains. The example here shows one potential method of addressing this problem.

The solution demonstrated *Section: 10.21.1 Simulink Code for Synchronization of Dexterit-E Data with an External Clock Example* uses the Sync Pulse Generator block to create a set of signals that an external system can make use of. The details of the signals can be found in *Section: 11.3 Kinarm I/O*.

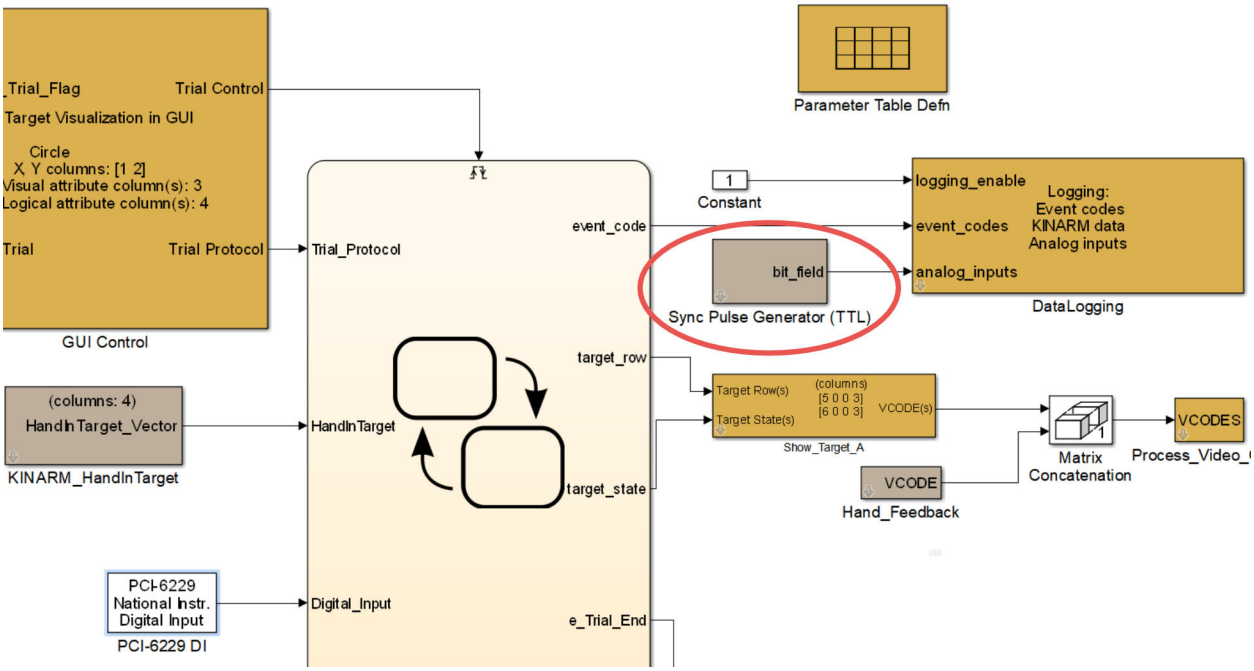
10.21.1 Simulink Code for Synchronization of Dexterit-E Data with an External Clock Example

In this example, which is based upon the ideas presented in *Section: 10.20 Digital Input/Output*, the `Sync Pulse Generator block` is used.

The `Sync Pulse Generator` block makes use of the NI-PCI-6229 block to output a set of digital pulses. See *Section: 11.3 Kinarm I/O* for a complete description of the generated signals. Within the Simulink model, the `Sync Pulse Generator` block's output needs to be saved as an `analog_input` via the `Datalogging` block, so that it can be used later to synchronize data.

If your Kinarm Lab does not contain an NI-PCI-6229 card then it is possible to unlink the Sync Pulse Generator from the library and replace the 6229 digital output block with a block that matches the hardware in your Kinarm Lab.

Figure 10-31: Sync Pulse Generator Simulink example

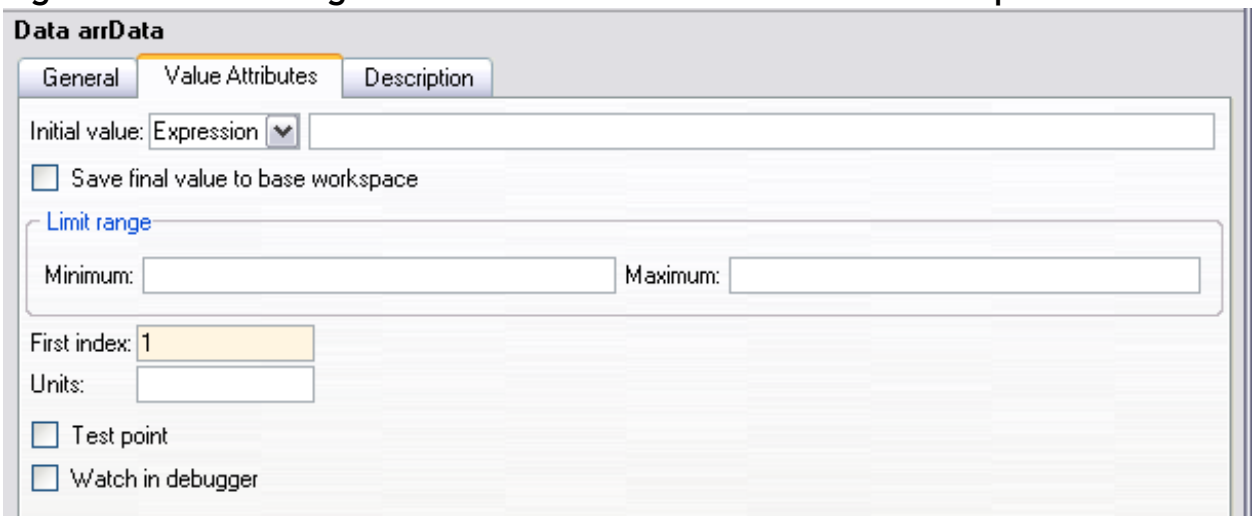


10.22 Using Vectors in Stateflow

If you wish to use a vector in your Stateflow chart it is essential that if it is a local variable that you set the vector's size. In addition, if you are using C as the Action Language for Stateflow*Section: 14.17 Update the Stateflow Action Language to MATLAB*, then you must also set its starting index. Users familiar with MATLAB will be well aware that MATLAB uses 1-based indexing.In contrast, C is nominally 0-based. If C is used as the Action Language then the starting index can be set within the Model Explorer. Select the vector you wish to work with and look at its properties. On the Value attributes tab be sure to set the First index field.

NOTE: The default Action Language for new Stateflow flow charts was C for MATLAB R2015a and earlier. Starting with R2019b, the default Action Language is MATLAB. See *Section: 14.17 Update the Stateflow Action Language to MATLAB* for tips on changing the Action Language for existing Stateflow charts

Figure 10-32: Defining the First Index of a Vector in the Model Explorer



10.23 Using Data from an Input File to Drive Exam Behaviour

This example demonstrates how to generate a binary file of data before a task is run and use that data to influence how your task behaves. In this example the hand paths from a previous exam are used to generate visuals. It would also be possible to extend this example to use the hand paths of a previous exam to guide the hand in a new exam.

The problems that need to be solved here are related to the starting and stopping of data reading as well as how to properly write and read the binary information.

When numbers are stored on a computer they are typically either in integer or floating point format. There is also the concept of byte-order or endianess when storing any type of number. Numbers can be big-endian or little-endian byte order. Without going into great detail on this topic, the default for the Robot Computer is little-endian, which means it is essential to write your data in little-endian format. All programming languages allow you to specify the endianess of the numerical data you write.

NOTE: If you plan to use the hand trajectories from a previous exam to drive the position controller in another exam it is essential to properly filter the data before writing it to a binary file. Without filtering the movement will be noisy at a level that will make the motion of the Kinarm jerky.

NOTE: The Input Files feature requires an accessible hard drive on the Robot Computer. Robot Computers with PN 13107 or higher will work with Input Files.

10.23.1 MATLAB Code for Creating an Input File

The following MATLAB code can be used to read data from a previous exam and use it to generate an Input File that can be used by a Task Program.

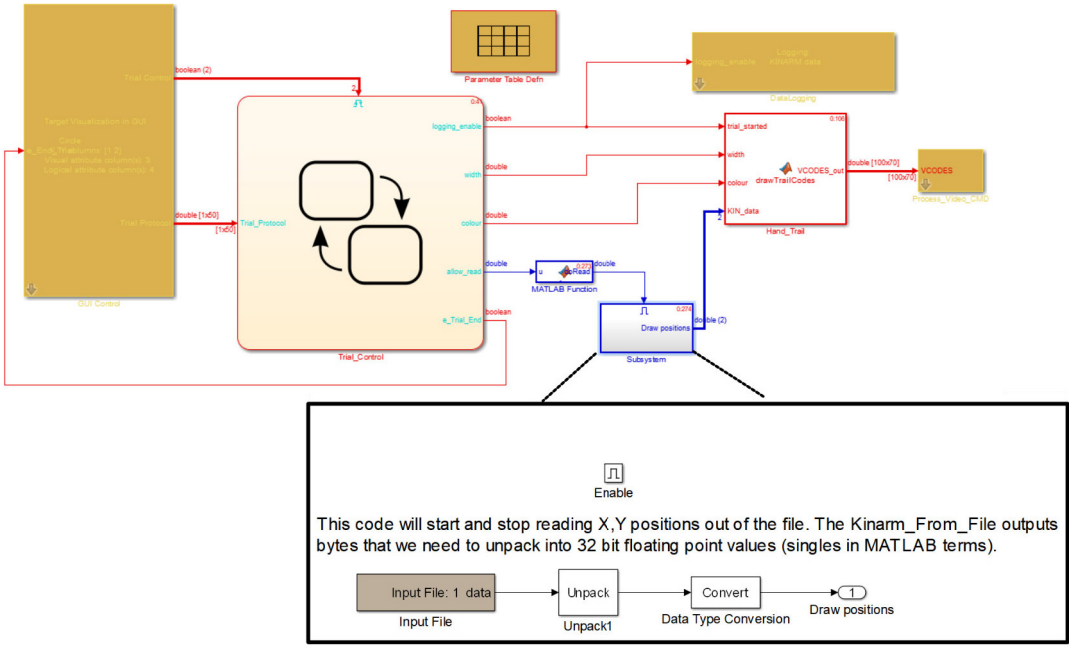
```
data = exam_load;
% Note: Filtering the position data is ESSENTIAL if you plan to feed
it into the
% position controller.
data = filter_double_pass(data, 'standard', 'fc', 10);
pathX = [];
pathY = [];

% Pull the hand position from each trial and combine into single
vector
for n=1:length(data.c3d)
 pathX = [pathX; data.c3d(n).Right_HandX];
 pathY = [pathY; data.c3d(n).Right_HandY];end

% Store the hand position as X1,Y1, X2 Y2...
xy = [pathX pathY];
out = reshape(xy', length(pathX) * 2, 1);
fileID = fopen('path.dat', 'w');

% The following 'l' specifies to write the data in little-endian
format, which is
% essential for reading the data later.
fwrite(fileID, out, 'single', 'l');
fclose(fileID);
```

10.23.2 Simulink Code for Reading Input Files

The Simulink code for this example is based on a task (available on our website) that takes the hand paths from a previous exam and draws them in the current exam. The contents of the enabled subsystem are shown below. The Input File block is set to read File 1. File 1 will therefore need to be one of the inputs specified in Dexterit-E when creating the Task Protocol. See the Dexterit-E User Guide for how to add an Input File to a Task Protocol. In the example below, 8 bytes of data are read from the file at a time and then converted into 2 singles which are then converted to 2 doubles. The doubles are used to create the X,Y position of the hand.

The subsystem is an enabled subsystem so that the task can control when reading the input file starts and stops. Without the enabled subsystem, as soon as the task start running it will start reading the Input File.

Figure 10-33: Simulink code for the Input File block



10.24 Task Instructions

A task may have instructions that are displayed to the operator prior to running the task. The instructions may be defined for the Task Program and/or for the Task Protocol. Protocol level instructions are set in the protocol editing interface of Dexterit-E. If a protocol has instructions set then the Task Program instructions are not displayed, just the protocol instructions.

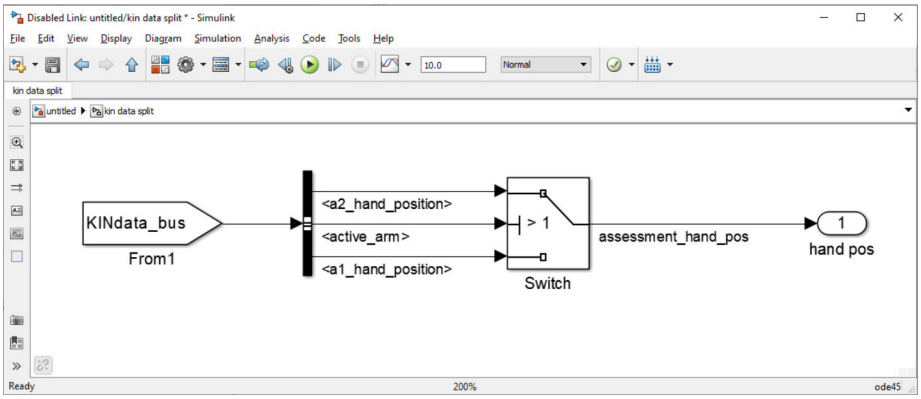
Default instructions for your Task Program can be added via the `Parameter Table Defn` block.

Task Protocols can specify video instructions to display to the subject on the subject display. For information on setting up video instructions see the Dexterit-E User Guide.

10.25 Using KINdata_bus on a Bilateral Lab

The `KINdata_bus From` tag contains most of the real-time kinematic required to dynamically control a Task Program. `KINdata_bus` contains named fields for all of the available kinematics. When creating a task that will run on either arm of a bi-lateral Kinarm Lab it is necessary to pull out fields for the correct arm from `KINdata_bus`. *Figure: 10-34* shows the procedure one can use to dynamically extract data from `KINdata_bus`. The idea is to use the active arm field and the `switch` block to determine whether arm 1 (right) data is output or arm 2 data (left).

Figure 10-34: Getting information from KINdata_bus

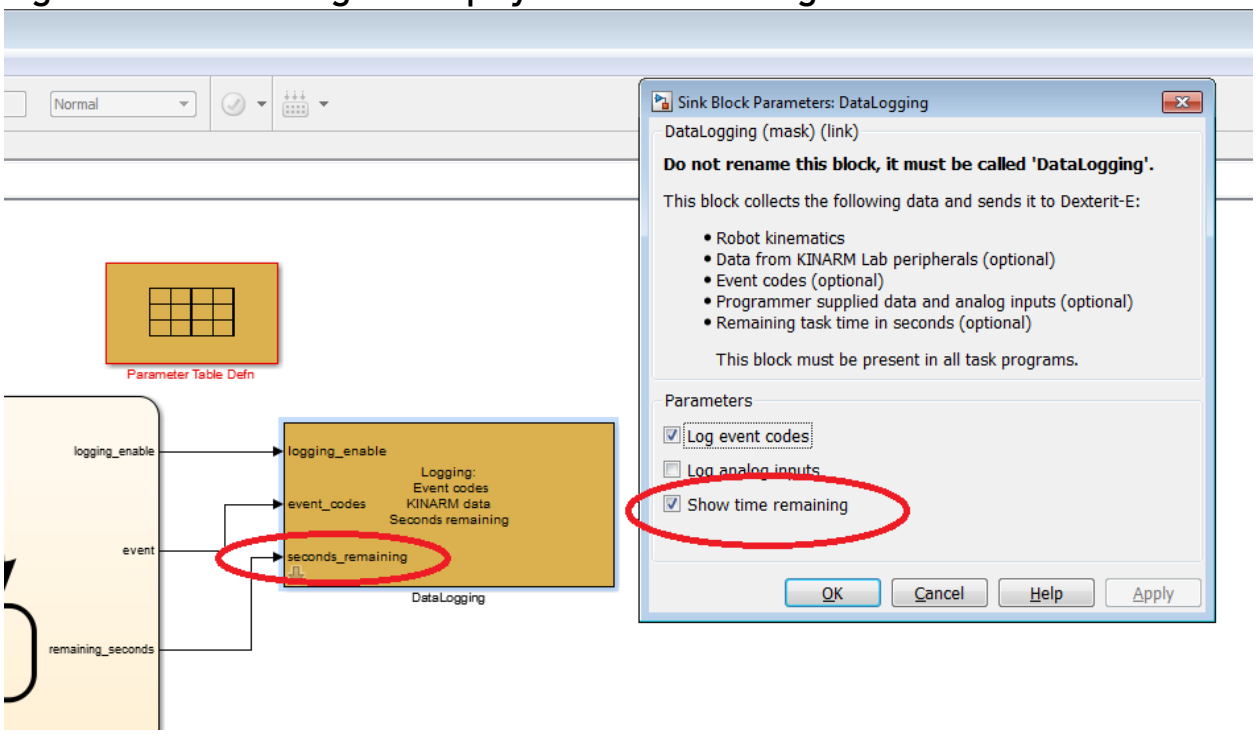


10.26 Show Time Remaining in a Task

If your task runs for a known (or calculable) amount of time, then it is possible to display the amount of time remaining in Dexterit-E (*Figure: 10-19*). The `Show time remaining` option in the `DataLogging` block creates an input on that block called `seconds_remaining`. Whatever value is passed into this input is displayed in Dexterit-E as the time remaining. Therefore, to behave correctly the new input must be passed the number of seconds remaining in your task (i.e. this value is not calculated automatically).

1. Check **Show time remaining** in the `DataLogging` block.

Figure 10-35: Enabling the Display of Time Remaining in the Task



2. Connect a signal that contains the number of seconds left in the task to this **Show time remaining** input.

   Whatever values is fed into this input will be displayed to the operator as part of the Task Status in the Dexterit-E UI.

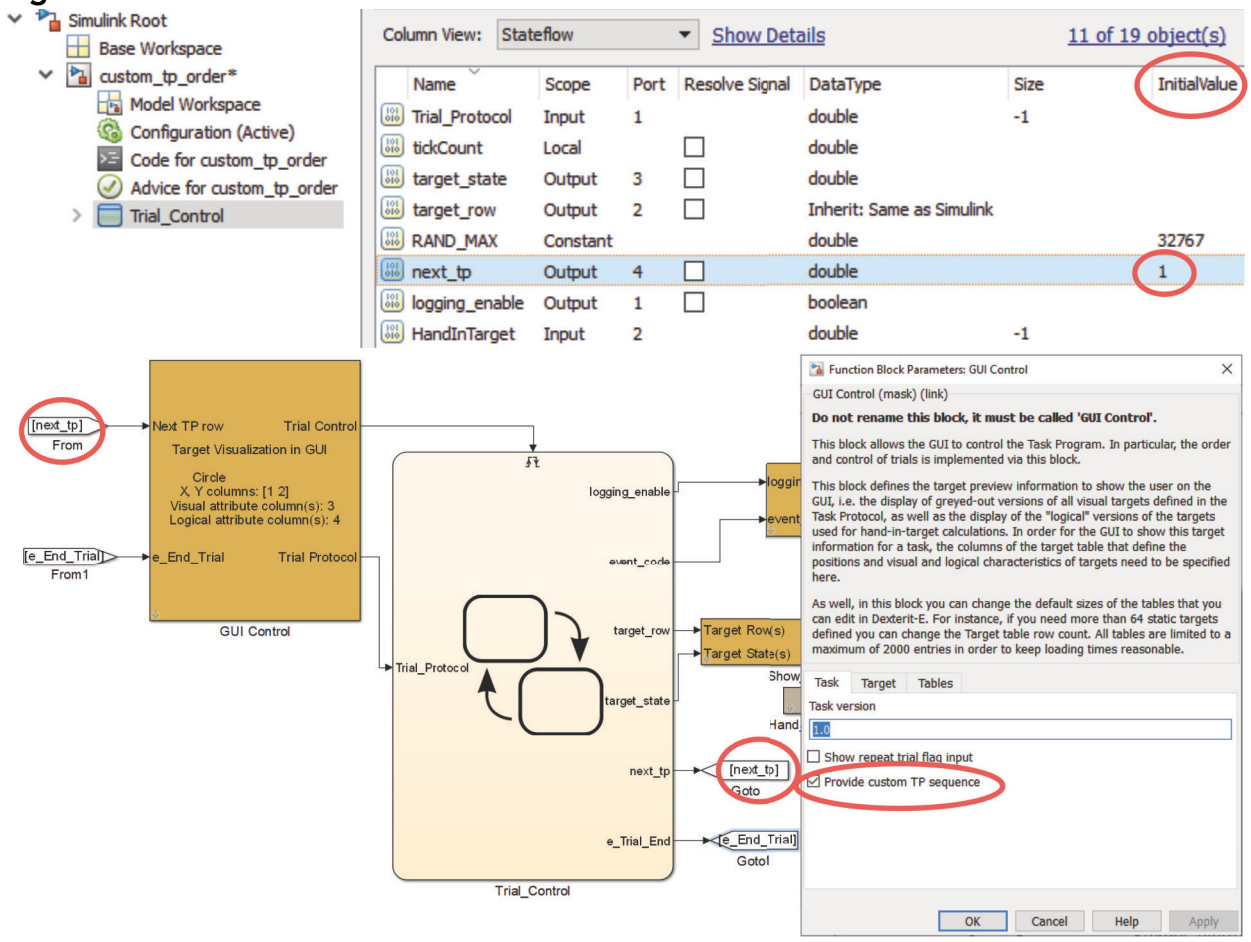
10.27 Custom Control of the Trial Protocol Order

Normally, Dexterit-E manages and controls the order in which Trial Protocols (TPs) execute based on what is defined in the block table of the Task Protocol (see the Dexterit-E User Guide for more details). However, if your task requires online decision making to determine the next TP to run, then it is possible to modify your task program so that it can have full control of TP execution order.

There are some important caveats and usage notes for custom TP ordering. These are, as follows:

- If you would like to use your existing `Trial_Control` Stateflow block to control the TP sequence, make sure the initial value for the first TP is set before the first Stateflow state in `Trial_Control` is entered. The `Trial_Control` block does not enter its first Stateflow state until the task starts; the `GUI Control` block needs the ID of the first TP before this. In practice, this is addressed by setting the initial value of the Stateflow output variable for the `Next TP row` using the Model Explorer.
- The task will stop immediately if the TP value entered via `Next TP row` is either less than or equal to zero or is larger than the size of your table.
- The `Next TP row` input must be updated/specified at least one kinematic frame before the `e_End_Trial` signal is sent to the `GUI Control` block. When the `e_End_Trial` signal is received by the `GUI Control` block, the value of the `Next TP row` is used immediately to choose the next TP. In practice, set the `Next TP row` in one Stateflow state and the next Stateflow state can send the `e_End_trial` signal.

Figure 10-36: Custom Trial Protocol Order



To allow your task program to control the TP order:

1. Double-click on the `GUI Control` block.
2. Check **Provide custom TP** sequence.

   See *Figure: 10-36*.

   This option creates a new input on the `GUI Control` block called `Next TP row`. The block table in the Task Protocol is ignored.

11 Dexterit-E Task Development Kit Libraries

All of the Simulink blocks contained within the various libraries provided as part of Dexterit-E Task Development Kit have built-in help files to describe in detail how the block should be used (help files are accessed through Simulink). Those help files are not replicated here, but rather a broad description of each of the libraries is provided to understand the organization of the various blocks.

Sometimes it may be helpful to make changes to one of these blocks in your model. To make changes to a block in your model you can right-click on the block and select **Link options** -> **Disable link**. The next time you compile the unlinked block will turn green so that you know you have made changes to the library block.

11.1 General

This library contains general blocks that are required for use with Task Programs:

- `GUI Control` - required for communications with the Dexterit-E GUI.
- `DataLogging` - required for communications with the Dexterit-E GUI.
- `Parameter Table Defn` - required for the column names and types for protocol creation with the Dexterit-E GUI.

11.2 Kinarm General

This library contains general blocks related to the Kinarm Lab:

- `Hand_Feedback` - required if hand feedback display to the subject is desired.
- `Gaze_Feedback` - required if gaze feedback display to the subject is desired, only works on systems with a Kinarm Gaze-Tracker installed.
- `HandInTarget` - provides feedback indicating which targets in the Target Table one of the hands is in.
- `DistanceFromTarget` - provides feedback indicating the distance between a targets and one of the hands.
- `GazeInTarget` - provides feedback indicating which targets in the Target Table that the subject's gaze is in. This block in only applicable to systems with a Kinarm Gaze-Tracker.

11.3 Kinarm I/O

This library contains input/output blocks related to the Kinarm Lab (e.g. motion control card, analog and digital I/O):

- `Analog Inputs` - retrieves specified analog inputs from a NI PCI-6071E or NI PCI-6229, auto-detecting which of those boards is present in a system. It outputs zeros in place of the retrieved analog input values if neither board is found.
- `Robot Digital Input` - retrieves the state of the digital inputs available on an Kinarm Labs with integrated digital inputs.
- `Robot Digital Output` - accepts 0's and 1's as inputs to send to the digital outputs available on Kinarm Labs with integrated digital outputs.
- `External_DAQ-Analog_Outputs` - outputs Kinarm robot kinematic information over analog channels for recording by an external data acquisition system.
- `External_DAQ-Digital_Outputs` - outputs two digital signals over cables labeled "Reward" and "Record" - typical use is for the reward signal and to trigger an external system to begin recording (e.g. eye-tracking system), but they can be used for anything.
- `Internal_DAQ-clock_Output` - outputs 1 kHz clock from the Robot Computer over a digital channel when Dexterit-E is performing data acquisition.
- `Plexon_Digital_Event` - outputs Task Event Codes to an external DAQ system, such as one manufactured by Plexon, and also controls when it records data.
- `Sync Pulse Generator (TTL)` - Uses the digital outputs on the NI-6229 card (if present) that can be used to synchronize Kinarm data with external systems. The output of this block is a bit field composed of DO3-7. This output should be saved as a analog channel via the `Datalogging` block.
  - **DO2** - Strobe pulse. Produces a 0.5 ms pulse, 0.5 ms after a change on ANY other DO bit. This output can be used to strobe/trigger an external data bus that only records when a strobe signal is sent.
  - **DO3** - Clock. Pulse train from the time the model starts: low for 10 ms, high for 5 ms
  - **DO4** - Task running. Outputs high as long as the task is running (i.e. the play button has been pressed)
  - **DO5** - Recording. Outputs high as long as Dexterit-E is recording data to an exam file. Some parts of some exams may not be recorded (e.g., reaches back to a center location).
  - **DO6** - Trial start. When a new trial starts this goes high for 50 ms.
  - **DO7** - Trial count (serial). The number of trials run during the task. This output produces a 10 bit digital number serially. It is high for 35 ms, followed by each of the 10 bits (LSB first) in synchrony with the clock pulses (DO3)
- National Instruments PCI-6229 blocks
  - `PCI-6229 AD` - Reads analog inputs from the card.
  - `PCI-6229 DI` - Reads digital inputs from the card.

  - `PCI 6229 PFI DI` - Reads digital inputs from the PFI inputs on the card.
  - `PCI-6229 DA` - Writes analog outputs to the card.
  - `PCI-6229 DO` - write digital outputs to the card.
  - `PCI 6229 PFI DO` - writes to the PFI digital outputs on the card.
- National Instruments PCI-6071E blocks
  - `PCI-6071E AD` - Reads analog inputs from the card
  - `PCI-6071E DI` - Reads digital inputs from the card
  - `PCI-6071E DA` - Writes analog outputs to the card
  - `PCI-6071E DO` - Writes digital outputs to the card
- National Instruments PCI-6503 blocks
  - `PCI-6503 DI` - Reads digital inputs from the card.
  - `PCI-603 DO` - Writes digital outputs to the card.
- National Instruments PCI-6713 blocks
  - `PCI-6713 DI` - Reads digital inputs from the card.
  - `PCI-6713 DA` - Writes analog outputs to the card.
  - `PCI-6713 DO` - Writes digital outputs to the card.
- UEI PD2-AO-8/16 blocks
  - `PD2-AO-8/16 Digital Input` - Reads digital signals from the card.
  - `PD2-AO-8/16 Analog Output` - Writes analog signals to the card.
  - `PD2-AO-8/16 Digital Output` - Writes digital signals to the card.
- UEI PD2-DIO-64 blocks
  - `PD2-DIO-64 Digital Input`- Reads digital inputs from the card.
  - `PD2-DIO-64 Digital Output` - Writes digital outputs to the card.

NOTE: MATLAB R2015a has its own version of the PCI card blocks. It is recommended that you use these versions for forward compatibility with newer MATLAB versions which do not include these blocks.

11.4 Kinarm Exo Loads

This library contains blocks for applying loads with a Kinarm Exoskeleton robot:

- `KINARM_Exo_Apply_Loads` - required for applying a load to the Kinarm robot.
- `KINARM_Exo_Position_Controller` - creates loads on the Kinarm that implement position control of the hand (finger-tip) or joint angles using a feed-forward model and feedback control.
- `Constant_Loads_(hand)` - applies a constant load in a hand-based coordinate frame.
- `Constant_Loads_(joints)` - applies a constant load in joint-based coordinate frame.
- `Velocity_Loads_(hand)` - applies velocity-based loads in a hand-based coordinate frame.
- `Velocity_Loads_(joints)` - applies velocity-based loads in joint-based coordinate frame.
- `Velocity_Loads_(hand, inter-arm)` - applies velocity-based loads in a hand-based coordinate frame, where load on one arm depends on velocity of other arm.
- `Velocity_Loads_(joints, inter-arm)` - applies velocity-based loads in joint-based coordinate frame, where load on one arm depends on velocity of other arm.
- `Forces_to_Torques` - converts forces in a hand-based coordinate frame to torques in a joint-based coordinate frame.
- `Perturbation` - creates a time-dependent profile that can be used as the scaling input to another load block to create a perturbation. A Stateflow event must be passed to the `e_Trigger` input on the block in order to start the scaling output changing from 0 to 1. The `e_reset` input on the block is also a Stateflow event that can optionally be used to immediately change the scaling output to 0.

11.5 Kinarm EP Loads

This library contains blocks for applying loads with the Kinarm Endpoint robot:

- `KINARM_EP_Apply_Loads` - required for applying a load to the Kinarm robot.
- `KINARM_EP_Position_Controller` - creates loads on the Kinarm End-Point robot that implement position control of the hand using a feed-forward model and feedback control.
- `Constant_Loads_EP_(hand)` - applies a constant load in a hand-based coordinate frame.
- `Perturbation_EP` - creates a time-dependent profile that can be used as the `scaling` input to another load block to create a perturbation. A Stateflow event is passed to the `e_Trigger` input on the block in order to start the scaling output

changing from 0 to 1. The `e_reset` input on the block is also a Stateflow event that can optionally be used to immediately change the scaling output to 0.

- `Velocity_Loads_EP_(hand)` - applies velocity-based loads in a hand-based coordinate frame.
- `Velocity_Loads_EP_(hand, inter-arm)` - applies velocity-based loads in a hand-based coordinate frame, where load on one arm depends on velocity of other arm.

11.6 Video

This library contains blocks used to display video:

- `Show_Target` - creates a `VCODE` containing all target information based on the Target Table and target selection. This block can take a single target ID or a vector of target IDs to show multiple targets.
- `Show_Target_With_Label` - creates a `VCODE` containing all target information based on the Target Table and target selection for targets with text. This block can take a single target ID or a vector of target IDs to show multiple targets.
- `Set_Target_in_Background` - takes a `VCODE` and turns it into a target definition that does not need to constantly be transmitted once its state is set.

  NOTE: This block is considered deprecated and will be removed in a future release.

- `Process_Video_CMD` - processes `VCODES` and communicates with the Dexterit-E Computer which actually displays the video.

12 Custom Simulink Blocks and Libraries

In many cases, you may want to create your own custom Simulink block to implement a novel feature. Furthermore, you may want such a block to exist in its own custom library to make it available to multiple Task Programs. This chapter provides basic instructions on creating custom Simulink blocks and adding them to a custom Simulink library.

12.1 Create a Custom Simulink Block

To create your own customized Simulink blocks, we recommend that you begin by looking at the blocks provided in the libraries as part of Dexterit-E. Many of the blocks provided with Dexterit-E include all of the Simulink and MATLAB source code which provides an excellent 'tutorial' for how to program. To see this source code:

1. Drag a block from one of Dexterit-E's Simulink libraries to a Task Program.
   Make sure to choose a block that is similar to what you are intending to do with your custom block.
2. Right-click on the block and under **Link Options** choose **Disable Link**.
   This disables the link between the library and this instance of the block.
3. Right-click on the block and choose **Look Under Mask**. This opens the Simulink block.

12.2 Put Your Custom Block Into a Custom Library

If you intend on using a custom block in more than one Task Program, it is advantageous to put that custom block into a Simulink library.

If you want only your custom block to show up in the library (and not any of the sub-blocks within your custom block), then you will need to Mask your block (see MathWorks' documentation).

If you want that library to have sub-libraries, add subsystems to the main custom library and enter your custom blocks into the subsystems.

If you want your custom library to have sub-libraries that are separate library files (as per Dexterit-E's library structure) rather than contained within sub-systems, then the sub-systems in the main library should be empty, and then in each of the subsystems' `Block_Parameters`, go to `CallBacks` and the `OpenFcn`: add in the name of the sub-library here (should be in the same directory as the main library).

The following steps will enable you to create and see your own custom library.

1. To create a library, choose **File** -> **New** -> **Library** from within Simulink.
2. Save it somewhere appropriate with an appropriate name.
3. To have your custom library show up in the Simulink Library Browser, you need an `slblocks.m` file in the same directory that your custom library was just saved in.

   Select one from the Simulink directory and modify it as per instructions in the `slblocks.m` file.
4. Add the path of the directory containing the new library to the MATLAB Path (note: you will also have to close the Simulink Library Browser and restart it to have your new library show up).

13 Kinarm Analysis Scripts

A set of basic MATLAB scripts to load Kinarm data into MATLAB for custom analysis is provided on the Kinarm Support website.

13.1 Install Kinarm Analysis Scripts

1. Download the Kinarm Analysis scripts from the Kinarm Support website.
   These are found under the MATLAB section of Software Downloads page.
2. In Windows Explorer navigate to **Documents** -> **MATLAB**.
3. In the MATLAB folder create a new folder called **Kinarm Analysis Scripts**.
4. Unzip and copy the contents of the Kinarm Analysis Scripts ZIP file into the **Kinarm Analysis scripts** folder you just created.
5. In MATLAB select **Set Path**.
6. Add the **Kinarm Analysis Scripts** folder you created to the MATLAB path.

14 Reference

This chapter contains reference information that is useful for creating Task Programs.

14.1 Structure of KINdata and KINdata_bus

As of Dexterit-E 3.6 there are now two `From` tags that contain kinematic and related data of the Kinarms: KINdata and KINdata_bus. For backwards compatibility, the KINdata structure is described here in this section. however, it will become obsolete in a future version of the TDK and so it is not recommended for use in new tasks. Instead, the KINdata_bus should be used. The bus version of the `From` tag can be used with a Bus Selector block to pull out named signals of data. This new approach is recommended because it is easier and more robust than using the KINdata structure directly. In addition, any new parameters added in future versions of Dexterit-E will only be added to KINdata_bus, not in the original KINdata structure.

The read-only KINdata matrix, which can be accessed as shown in *Section: 10.16 Accessing KINdata_bus, Current Block Index, and Other Task Control Variables*, contains all kinematic data related to the Kinarm robot. It allows end-users to create custom effects based on feedback from the Kinarm robot. Within the KINdata matrix, row 1 contains data about the first arm and row 2 contains data about the second arm, as shown in the table *Table: 14-1* below. For bilateral Kinarm Labs, the first arm is the right arm and the second arm is the left arm. For unilateral systems, the first arm can be either the right or left arm, depending upon the Kinarm configuration.

Table 14-1: Arm specific data in KINdata matrix (i.e. KINdata(i,j), where i=1 or i=2) (Sheet 1 of 3)

j	Name	Description
1	L1	Length (m) of Kinarm segment L1, the upper arm (shoulder to calibrated elbow position)
2	L2	Length (m) of Kinarm segment L2, the forearm (calibrated elbow position to calibrated fingertip position)
3	L2_ptr	Location (m) of fingertip, relative to Kinarm segment 2. (Measured in the anterior direction when the arm is in the anatomical position, i.e. when the arm is straight out to the side)
4	SHO_X	Shoulder position (m), x-axis in global coordinate scheme.
5	SHO_Y	Shoulder position (m), y-axis in global coordinate scheme.
6	ORIENTATION	1= right-handed Kinarm robot, 2 = left-handed Kinarm robot
7	SHO_ANG	Shoulder angle (rad) in local coordinates (flexion is positive)
8	ELB_ANG	Elbow angle (rad) in local coordinates (flexion is positive)
9	SHO_VEL	Shoulder angular velocity (rad/s) in local coordinates
10	ELB_VEL	Elbow angular velocity (rad/s) in local coordinates
11	SHO_ACC	Shoulder angular acceleration (rad/s2) in local coordinates
12	ELB_ACC	Elbow angular acceleration (rad/s2) in local coordinates
13	SHO_TOR_CMD	Commanded shoulder torque (N·m) in local coordinates
14	ELB_TOR_CMD	Commanded elbow torque (N·m) in local coordinates
15	M1_TOR_CMD	Commanded M1 torque (N·m) in global coordinates (torque on segment L1 produced by motor M1)
16	M2_TOR_CMD	Commanded M2 torque (N·m) in global coordinates (torque on segment L2 produced by motor M2)
17	L1_ANG	L1 angle (rad) in global coordinates
18	L2_ANG	L2 angle (rad) in global coordinates
19	L1_VEL	L1 angular velocity (rad/s) in global coordinates
20	L2_VEL	L2 angular velocity (rad/s) in global coordinates
2	L1_ACC	L1 angular acceleration (rad/s2) in global coordinates
22	L2_ACC	L2 angular acceleration (rad/s2) in global coordinates

Table 14-1: Arm specific data in KINdata matrix (i.e. KINdata(i,j), where i=1 or i=2) (Continued) (Sheet 2 of 3)

j	Name	Description
23	HPX	Position of calibrated fingertip (m), x component, global coordinates
24	HPY	Position of calibrated fingertip (m), y component, global coordinates
25	HVX	Velocity of calibrated fingertip (m/s), x component, global coordinates
26	HVY	Velocity of calibrated fingertip (m/s), y component, global coordinates
27	HAX	Acceleration of calibrated fingertip (m/s2), x component, global coordinates
28	HAY	Acceleration of calibrated fingertip (m/s2), y component, global coordinates
29	EPX	Position of calibrated elbow (m), x component, global coordinates
30	EPY	Position of calibrated elbow (m), y component, global coordinates
31	EVX	Velocity of calibrated elbow (m/s), x component, global coordinates
32	EVY	Velocity of calibrated elbow (m/s), y component, global coordinates
33	EAX	Acceleration of calibrated elbow (m/s2), x component, global coordinates
34	EAY	Acceleration of calibrated elbow (m/s2), y component, global coordinates
35	SHO_VEL_FLT	Shoulder angular velocity (rad/s) in local coordinates, filtered with 2nd order 10 Hz Butterworth filter
36	ELB_VEL_FLT	Elbow angular velocity (rad/s) in local coordinates, filtered with 2nd order 10 Hz Butterworth filter
37	MOT_STATUS	Motor status bit field
38	FS_FORCE_U	Force measured by force sensor (N), x component, in local coordinates
39	FS_FORCE_V	Force measured by force sensor (N), y component, in local coordinates

Table 14-1: Arm specific data in KINdata matrix (i.e. KINdata(i,j), where i=1 or i=2) (Continued) (Sheet 3 of 3)

j	Name	Description
40	FS_FORCE_W	Force measured by force sensor (N), z component, in local coordinates
41	FS_TORQUE_U	Torque measured by force sensor (N·m), x component, in local coordinates
42	FS_TORQUE_V	Torque measured by force sensor (N·m), y component, in local coordinates
43	FS_TORQUE_W	Torque measured by force sensor (N·m), z component, in local coordinates
44	FS_FORCE_X	Force measured by force sensor (N), x component, in global coordinates
45	FS_FORCE_Y	Force measured by force sensor (N), y component, in global coordinates
46	FS_FORCE_Z	Force measured by force sensor (N), z component, in global coordinates
47	FS_TORQUE_X	Torque measured by force sensor (N·m), x component, in global coordinates
48	FS_TORQUE_Y	Torque measured by force sensor (N·m), y component, in global coordinates
49	FS_TORQUE_Z	Torque measured by force sensor (N·m), z component, in global coordinates
50	FS_TIMESTAMP	The time stamp from the torque sensor (s).

NOTE: All angular positions, velocities, and accelerations are defined around the proximal joint for each segment. Segment definitions can be found in *Section: 14.2 Kinarm Segment Definitions*.

Within the KINdata matrix, row 3 contains data that is not arm-specific, as shown in table *Table: 14-2*.

Table 14-2: Non-arm-specific data in KINdata matrix (i.e. KINdata(3,j))

j	Name	Description
1	active_arm	This is the arm selected in Dexterit-E as the **Arm to be assessed**. (1 = 1st arm, 2 = 2nd arm). Prior to Dexterit-E 3.6, the Arm to be assessed was referred to as the **Active arm**.
2	Feed_forward	Duration (s) of feed-forward estimate used for display of hand feedback (i.e. fingertip dot)
3	FDBK_STTS	Feedback status (0=none, 1=arm to be assessed, 2=contralateral arm, 3=both, 4=controlled by Task Program).
4	FDBK_SRC	Source of feedback style, colour etc. (0=from Dexterit-E GUI, 1=from Task Program)
5	FDBK_CLR	Feedback colour (RRRGGGBBB)
6	FDBK_SIZE	Radius of feedback dot (m)
7	ENBL_LOADS	Status of load enable from Dexterit-E
8	VEL_DLY_PRI	Delay of velocity signals (s) from primary encoders. Delay is produced by motion control card on-board calculations.
9	ACC_DLY_PRI	Delay of acceleration signals (s) from primary encoders. Delay is produced by motion control card on-board calculations.
10	VEL_DLY_SEC	Delay of velocity signals (s) from optional secondary encoders.
11	ACC_DLY_SEC	Delay of acceleration signals (s) from optional secondary encoders.
12	SERVO_UPDATE	A counter from the servo amp
13	EP_CALIBRATION_BUTTON	On EP systems this is the state of the EP calibration button
14	GAZE_AVAILABLE	One if the system is registered as having gaze tracking, zero otherwise.
15	GAZE_FEEDBACK	Zero if gaze feedback to subject is on, one if gaze feedback is program controlled.

Table 14-3: KINdata_bus variables (Sheet 1 of 4)

Section	Name	Definition
KINDataGeneral	active_arm	Deprecated. Use robot_arm and subject_arm.
	delay_estimates	Four element array with the delay in seconds for: 1=primary encoder velocity measure, 2=primary encoder acceleration measure, 3=secondary encoder velocity measure, 4=secondary encoder acceleration measure.
	servo_counter	The update counter from the servo.
	calibration_button_bits	Bit field indicating which calibration buttons have been touched (first bit is robot 1, second bit is robot 2).
	robot_arm	This is the arm selected in Dexterit-E as the **Arm to be assessed**. (1 = right arm, 2 = left arm). However, if you have a unilateral Kinarm End-Point robot this will always be 1.
	subject_arm	This is the arm selected in Dexterit-E as the **Arm to be assessed**. (1 = right arm, 2 = left arm)
Hand_feedback	handFB_feed_forward_time	The time that is used to calculate the position of the hand feed-back (s; typically 0.050 s).
	handFF_DEX	The time that Dexterit-E requests for handFB_feed_forward_time (s).
	handFB_arm	The robot(s) for which hand feedback should be provided. 0 = no hand feedback, 1 = assessment arm, 2 = non-assessment arm, 3 = both arms.
	handFB_radius	The default radius for the hand feedback circle.
	handFB_control	Determines whether Dextertit-E controls the hand feedback status (<> 4) or if the task program controls the hand feedback (4).
	handFB_color	The colour of the hand feedback circle.
Robot1 / Robot2	link_lengths	Length (m) of Kinarm segment L1, the upper arm (shoulder to calibrated elbow position) and length (m) of Kinarm segment L2, the forearm (calibrated elbow position to calibrated fingertip position).

Table 14-3: KINdata_bus variables (Continued) (Sheet 2 of 4)

Section	Name	Definition
	pointer_offset	Location (m) of fingertip, relative to Kinarm segment 2. (Measured in the anterior direction when the arm is in the anatomical position, i.e. when the arm is straight out to the side).
	shoulder_position	(X, Y) shoulder position in global coordinates (m).
	arm_orientation	1= right-handed Kinarm robot, 2 = left-handed Kinarm robot.
	shoulder_angle	Shoulder angle (rad) in local coordinates (flexion is positive).
	elbow_angle	Elbow angle (rad) in local coordinates (flexion is positive).
	shoulder_ang_velocity	Shoulder angular velocity (rad/s) in local coordinates.
	elbow_ang_velocity	Elbow angular velocity (rad/s) in local coordinates
	shoulder_ang_acceleration	Shoulder angular acceleration (rad/s2) in local coordinates.
	elbow_ang_acceleration	Elbow angular acceleration (rad/s2) in local coordinates
	joint_torque_command	Commanded shoulder and elbow torque (N·m) in local coordinates.
	motor_torque_command	Commanded M1 and M2 torques (N·m) in global coordinates (torque on segment L1 produced by motor M1, and torque on segment L2 produced by motor M2).
	link_angles	L1 and L2 angles (rad) in global coordinates.
	link_velocities	L1 and L2 angular velocities (rad/s) in global coordinates.
	link_accelerations	L1 and L2 angular accelerations (rad/s2) in global coordinates.
	hand_positions	X, Y position of calibrated fingertip / handle (m), global coordinates
	hand_velocities	X,Y velocities of calibrated fingertip / handle (m/s), global coordinates

Table 14-3: KINdata_bus variables (Continued) (Sheet 3 of 4)

Section	Name	Definition
	hand_accelerations	X, Y acceleration of calibrated fingertip / handle (m/s2), global coordinates
	elbow_position	X, Y position of the elbow (m), global coordinates
	elbow_cart_velocity	X,Y velocities of the elbow (m/s), global coordinates
	elbow_cart_acceleration	X, Y acceleration of the elbow (m/s2), global coordinates
	motor_status	Motor status bit field.
	force_sensor_force_uvw	Force measured by force sensor (N), in local coordinates.
	force_sensor_torque_uvw	Torque measured by force sensor (N·m), in local coordinates.
	force_sensor_force_xyz	Force measured by force sensor (N), in global coordinates.
	force_sensor_torque_xyz	Torque measured by force sensor (N·m), in global coordinates.
	force_sensor_timestamp	The time stamp from the force sensor (s).
	force_sensor_status	Status bit field from the force sensor.
	ecatRecordedMotorTorques	The recorded torques applied to the motors in EtherCAT controlled systems (arm 1 M1, M2, arm 2 M1, M2).
	ecatCurrentLimitEnabled	A bit field indicating which motors have current limits enabled. Bit order is: arm 1 M1, M2, arm 2 M1, M2.
	ep_grip_sensor	A number indicating if the grip sensor on a Kinarm End-Point robot is held (1) or not (0). This field is only valid on Kinarm End-Point robots, not the Kinarm End-Point Classic robot.
ECATDigitalInput	n/a	An array of 8 values, one for each digital input on a bilateral Kinarm robot controlled using EtherCAT.
gaze	has_gaze	A boolean indicating whether or not the system had a Kinarm Gaze Tracker.

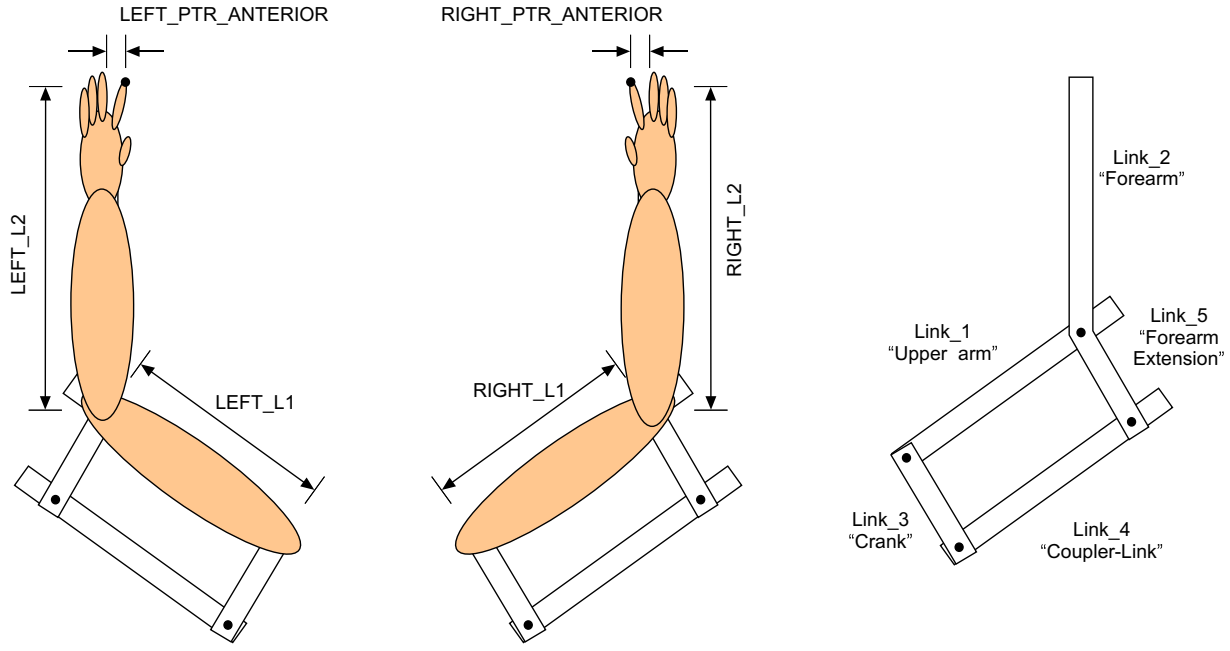
Table 14-3: KINdata_bus variables (Continued) (Sheet 4 of 4)

Section	Name	Definition
	gazeFB_control	Similar to handFB_control. 0 = off, 1 = on, 2 = task program control.
	gaze_location	The location of the subject's gaze in global coordinates (m).
	gaze_timestamp	The timestamp from the Kinarm Gaze Tracker (s).
	gaze_pupil_area	The area of the pupil (m2).
	gaze_event	A Kinarm Gaze Tracker event code. See Table 10-5: "Gaze tracking data matrix".
	gaze_vector	A 3D vector indicating the direction of the subject's gaze (unit-less).
	gaze_pupil_loc	The 3D location of the pupil in global coordinates (m).

14.2 Kinarm Segment Definitions

The various segments of the Kinarm robot, as well as the subject arm lengths that are calibrated during subject setup with a Kinarm Exoskeleton robot, are shown from a top-down view in *Figure: 14-1*. The segments of the Kinarm End-Point robot follow the same numbering scheme.

Figure 14-1: Kinarm Segment Definitions



14.3 Force Plate Data

Kinarm End-Point Labs with the Adjustable Height Configuration can come with one or two Kinarm Lab force plates. In order to use the force plate data in real time in your Simulink model you need to make use of the Simulink `From` tag (similar to accessing the KINdata structure). The specific `From` tag to use is `ForcePlateData`. The `ForcePlateData` structure contains 14 entries:

Table 14-4: Force plate data matrix

Index	Name	Description
1	P1_FX	Force plate 1 force in global X direction (N).
2	P1_FY	Force plate 1 force in global Y direction (N).
3	P1_FZ	Force plate 1 force in global Z direction (N).
4	P1_MX	Force plate 1 moment of inertia in global X direction (N·m).
5	P1_MY	Force plate 1 moment of inertia in global Y direction (N·m).
6	P1_MZ	Force plate 1 moment of inertia in global Z direction (N·m).
7	P1_Timestamp	Relative time stamp from force plate 1 for the given measures (s).
8	P2_FX	Force plate 2 force in global X direction (N).
9	P2_FY	Force plate 2 force in global Y direction (N).
10	P2_FZ	Force plate 2 force in global Z direction (N).
11	P2_MX	Force plate 2 moment of inertia in global X direction (N·m).
12	P2_MY	Force plate 2 moment of inertia in global Y direction (N·m).
13	P2_MZ	Force plate 2 moment of inertia in global Z direction (N·m).
14	P2_Timestamp	Relative time stamp from force plate 2 for the given measures (s).

14.4 Kinarm Gaze-Tracker Data

Kinarm labs can come with a Kinarm Gaze-Tracker. To use the gaze tracking data in real-time in your Simulink model you need to make use of the Simulink `From` tag. This is similar to accessing the KINdata structure. *Table: 14-5*, below, shows the available `From` tags.

Table 14-5: Gaze-Tracking Data Matrix

`From` Tag	Index	Name	Description
`Gaze_Global`	1	X	The X position of the subject's gaze in global coordinates. (m)
	2	Y	The Y position of the subject's gaze in global coordinates. (m)
	3	Timestamp	The time stamp from the Kinarm Gaze-Tracker for the current gaze position data. (s)
`Gaze_Pupil_Area`	1	Pupil area	The area of the subject's pupil (m^2) if the Kinarm Gaze-Tracker is calibrated and running.
`Gaze_Event_Info`	1	Event ID	An integer defining the type of event: • 4 - Blink • 6 - Saccade • 8 - Fixation
	2	Start time	The time the event started relative to when the task loaded. (s) See note below.
	3	End time	The time the event ended relative to when the task loaded. (s) See note below.
`Gaze_Vector_Global`	1	X	The X component of a unit vector pointing from the subject's eye to the gaze location in the workspace.
	2	Y	The Y component of a unit vector pointing from the subject's eye to the gaze location in the workspace.
	3	Z	The Z component of a unit vector pointing from the subject's eye to the gaze location in the workspace.
`Gaze_Pupil_Global`	1	X	The X location of the subject's pupil in the global workspace. (m)
	2	Y	The Y location of the subject's pupil in the global workspace. (m)
	3	Z	The Z location of the subject's pupil in the global workspace. (m)

NOTE: Gaze events will always be received by the Robot Computer with a delay compared to the time they occurred. The delay is often at least 50ms after the end of the event. When accessing them using a `From` tag, the Timestamp will be relative

to when the task was loaded, which is slightly different to how they are saved to the data file which corrects the event time to the frame at which it occurred.

14.5 Stateflow Events versus Task Event Codes

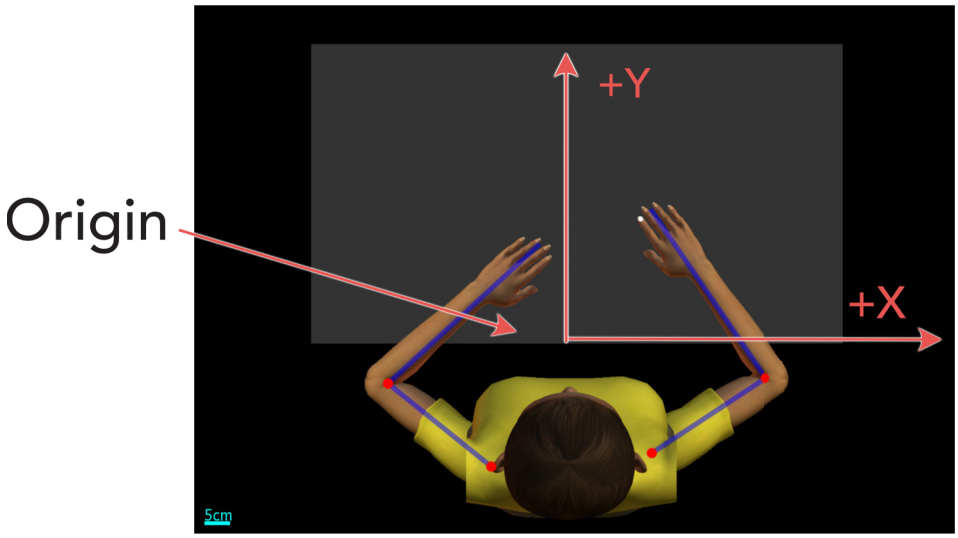
As mentioned previously in this document, Stateflow events should not be confused with Task Events Codes. Stateflow events are Stateflow objects that drive Stateflow diagram execution. They are not saved in a data file for subsequent data analysis. Task Event Codes are user-defined codes that have no effect on a Stateflow chart, but they are saved in a data file for subsequent data analysis as “events”.

14.6 Global Coordinate System

All data accessed in a Task Program is defined in a single, global coordinate system. These data include the Kinarm kinematic data (see *Section: 14.1 Structure of KINdata and KINdata_bus*) and all video data (e.g. VCodes and the Target Table).

The global coordinate system is defined with positive X to the right and positive Y away from the subject towards the robots. The origin (0,0) is defined as being on the subject display, centered in the X direction, closest to the headrest in the Y direction. *Figure: 14-2* shows where the origin is for most Kinarm Labs.

Figure 14-2: Global coordinates



There is a difference between the values in the Target Table viewed by an operator in Dexterit-E and the values in the Target Table as accessed by a Task Program. In Dexterit-E, the operator sees and defines the Target Table positions in a local coordinate system (e.g. relative to the fingertip when the arm is in a certain orientation). When a Task Protocol is chosen and downloaded, however, those positions get translated to the global coordinate system for the Task Program, so that they are in the same coordinate system as everything else. This translation is automatic.

For back-projection based Kinarm Labs the origin may not be in the position indicated in *Figure: 14-2*. Although the origin is always at the middle of the bottom of the computer generated image, this does not always correspond to the middle of the bottom of the image viewed by the subject for a back-projection system. Two factors can lead to the global origin not being at the middle of the bottom of the viewable image: incomplete image projection in a back-projection system and/or a mismatch in resolutions between the display device and Windows. Neither of these problems typically occurs with LCD monitors, whereas both often occur with project-based back-projection systems.

In a back-projection image system, the problem of incomplete image projection occurs if the screen is not large enough to display the entire projected image. If this situation occurs, then part of the projected image will be cut off and the origin may appear off-screen.

The other potential problem with projectors occurs when there is a mismatch in resolutions between the display device and Windows. More specifically, if the projector's aspect ratio setting is set to 'Native' and there is a mismatch between the projector's native resolution and the resolution chosen in the Windows Display Properties then part of the image will be cut off and the origin will not be at the middle of the bottom of the projected image. For example, consider a wide-screen projector with a native 1280 x 848 resolution and an aspect ratio setting set to 'Native'. Under these conditions, the projector will only display the first 1280 x 848 pixels that it receives. If the screen resolution in the Windows Display Properties is set to 1280 x 1024, which is larger than 1280 x 848, then part of the computer generated image will not be displayed. More specifically, only the first 848 lines of the computer generated image will be displayed. The remaining 176 lines (i.e., 1024 - 848 = 176) will not be displayed. So in this example the origin, which is part of the undisplayed image, will not be projected.

14.7 VCodes - Programming Visual Stimuli

In order for visual stimuli to be presented to the subject, one or more VCodes must be sent to the `Process_Video_CMD` block. A VCode is a 70 element vector where each element in the vector specifies some aspect of what will be drawn. Blocks like `Show_Target` and `Hand_Feedback` can be used to automatically create VCodes that are in a valid format. However, it can be useful or necessary to modify existing VCodes or create your own VCodes from scratch in Embedded MATLAB blocks.

General properties that a VCode can define are:

- Fill colour;
- Outline width;
- Outline colour;
- Opacity.

The shapes which can be drawn with a single VCode are:

- Circle - requires a radius;
- Ellipse - requires a major axis size, minor axis size and rotation;
- Rectangle - requires a height, width and rotation;
- Triangle - requires a height, width, peak offset, and rotation;
- Line segment - requires an end point offset;
- Path - requires up to 25 sequential locations;
- Polygon - requires up to 25 vertex locations.

14.7.1 Visual Display Limitations

When drawing to the subject display, timing is critical because each video frame must be drawn within less than the duration of a single video frame (i.e. 1/60th of a second when the subject display's refresh rate is 60 Hz). All graphics hardware has limitations on how much it can draw within a single frame. There are also limitations on how many VCodes can be transmitted at one time via the Dexterit-E LAN. As such, there are several limitations on VCodes:

- When images are used to draw targets some limits are imposed on the images.
  - Dexterit-E will only use 400 MB of video memory at a maximum. Images beyond the 400 MB limit will not be loaded.
  - Any image that requires more than 50 MB of video memory will not be loaded.

  NOTE: Video memory is not the same as the image's file size. The video memory an image will use is calculated as: Memory Size in MB = Image width in pixels * Image height in pixels * 4 bytes per pixel / (1024 * 1024).

- VCodes are transmitted on the Dexterit-E LAN at 1 kHz. As such, many video frames that are sent are never drawn.

  NOTE: In all versions of Dexterit-E before 3.8 VCodes were transmitted at 500 Hz.

- A maximum of 200 VCodes can be sent at a time to comprise one video frame.
- The number of VCodes that can be drawn each frame depends on several factors:
  - Shape: some shapes are more complex to draw. The more vertexes there are in the shape the more intensive the shape is to draw. For example. triangles are very simple to draw while polygons with 25 vertexes and large circles (which require many vertexes to look smooth) can reduce the potential maximum number of VCodes.

- Image fill: using images as the "fill colour" for a shape is more intensive than filling with a colour. Single large images (over 10 MB) may also affect drawing reliability.
- Computer: different models of the Dexterit-E Computer have different processing capabilities.
- For example, using Dexterit-E Computer PN 13107 or 13950 and MATLAB R2015a SP1, it is possible to draw:
  - 4800 lines, 5 mm thick (200 path VCodes, each with 25 vertices);
  - 75 circles, solid fill, radius of 2.5 cm (3 VCodes with 25 repeats each);
  - 500 circles, solid fill, radius of 1.0 cm (20 VCodes with 25 repeats each);
  - 200 rectangles, filled with JPG images, 10 x 10 cm (8 VCodes with 25 repeats each);
  - 75 text labels, 1 cm height, 2 characters each (75 VCodes).

If you are concerned that your task may be taking you close to the limit of video performance, then it is possible to verify the video performance of your task. In order to verify the video performance of your task you can either:

- Open an exam using Dexterit-E Explorer and review the **Video frame interval time** channel.
- Open an exam in MATLAB using `zip_load()` and find the VIDEO_LATENCY.ACK_TIMES fields in the structure. The difference between each entry in the ACK_TIMES vector indicates the video frame interval time in seconds.

These data indicate the time at which an acknowledgment that a frame was displayed was received by the Robot Computer. The dominant factor affecting this value is the actual inter-frame interval. Typically the recorded video frame interval times should be near 16.7 ms (i.e. the frame duration for a subject display with a 60 Hz refresh rate). However, the timing of the acknowledgments can be affected by other processes, so the recorded video frame intervals will have some amount of jitter between approximately 14 ms and 25 ms. If video frames are being dropped, then the video frame interval time will be close to some multiple of the frame duration. For example, if a video frame interval is close to 32 ms for a 60 Hz display, then one video frame was not drawn, and an interval of 49 ms means two frames were not drawn. For further questions regarding video performance of your task, contact **Kinarm Support**.

14.7.2 Overlapping Targets

When targets are drawn to the screen they are drawn in the order in which they are received. This means that if a matrix of targets is passed to the `Process_Video_CMD` block then `VCODE(1,:)` will be drawn before `VCODE(2,:)`. This ordering of VCodes means that later VCodes can draw over earlier VCodes, potentially hiding the earlier VCodes completely. If, for instance, you are finding that hand feedback is disappearing behind other targets then you will need to change the order your hand feedback is drawn in - the hand feedback would need to be drawn last.

14.7.3 Background Targets

Sometimes a task may have many targets that do not change frequently, but need to remain on the screen. It can be inefficient to constantly re-transmit those VCodes. It is possible to use background targets to solve this problem. Background targets are VCodes that contain an ID from 1-500 that is used to store the target so that it is always displayed without the need to re-send the VCode. Background targets need to be transmitted to Dexterit-E until they are accepted by Dexterit-E. The `Set_Target_in_Background` block is designed to help with determining when a target has been accepted. VCodes for background targets will always be drawn before other VCodes. This order of drawing means that background targets can be drawn over by VCodes being transmitted regularly.

Background targets need to have a unique ID. If you transmit another background target using an existing ID then the new VCode will overwrite the old VCode. For example, if you have sent a VCode for a background target with ID 23 as a red circle, then later send a VCode for a background target with ID 23 as a green rectangle, then the red circle is overwritten and only the green rectangle will be shown.

NOTE: The `Background Targets` block is considered a deprecated feature because it is now possible to send 200+ VCodes at a time. This feature will be removed in a future release.

14.7.4 Repeat Targets

At times it may be useful to display a shape with the same properties (size and colour) in many locations at one time. While this task can be accomplished using one VCode per target, it is more efficient to use a single repeat target VCode. A repeat target VCode specifies one set of parameters for the look of a target and also a series of locations for where the target is to appear. Repeat targets are more efficient in terms of network bandwidth and storage when dealing with many identical looking targets. Repeat targets can be created for circles, ellipses, rectangles, line segments, and triangles. The general idea is to fully specify the VCode for the given target type, then specify a number of repeats and an X,Y location in the global workspace for each repeat. See *Table: 14-6* for more details.

14.7.5 Specifying Colours

When specifying colours in VCodes there are a few rules to keep in mind:

- Values >= 0 are treated as RGB colours. Valid RGB colour values can be created using the create_color.m function provided in the Task Development Kit. For example, at the MATLAB prompt you can type `create_color(100, 0, 0)` and that will return the appropriate number to represent red in a VCode.
- colour values for the fill colour of a VCode can be <0, these are interpreted as image indexes. -1 means the first image alphabetically by file name that is found in the current task's folder, -2 is the second image alphabetically, etc.
- Images cannot be used for a stroke colour.

14.7.6 VCode Format

The first table in this topic displays each of the 70 indices used to describe the vector elements used when drawing stimuli. The second table describes the first index in more detail.

Table 14-6: VCode Vector Format (Sheet 1 of 3)

Index	Valid Values	Description
1	Various values within the range of 1 to 125. See *Table: 14-7* for a list.	1 - Circle 2 - Ellipse 3 - Rectangle 4 - Line Segment 5 - Triangle 6 - Path 7 - Polygon Target with text - Add 10 to any of the numbers listed above if you want to show text with the target. For example, 13 is a rectangle with text. However, 16 and 17 are not valid as text cannot be shown with a path or a polygon. Repeat targets - Add 20 to any of the numbers listed above if you want to have repeats of the same target. For example, 25 is the same triangle drawn in many locations. However, 26 and 27 are not valid as you cannot repeat paths or polygons. Background targets - Add 100 to any of the numbers listed above if you want to set the target in the background. To set a target in the background means that when you stop sending the VCode, the target continues to display. Normally, this is accomplished with the `Set_Target_in_Background` block. For example, 121 adds a circle with multiple repeats to the background.
2	0 to 3	Describes where to show the target 0 - Do not show the target 1 - Show the target on the subject and operator displays 2 - Show the target on the subject display only 3 - Show the target on the operator display only
3	-10 to 10	The X location of the target centre in meters in global coordinates. For line segment targets this is the location of the start of the line. For path targets and repeated targets this value is unused.
4	-10 to 10	The Y location of the target centre in meters in global coordinates. For line segment targets this is the location of the start of the line. For path targets and repeated targets this value is unused.

Table 14-6: VCode Vector Format (Continued) (Sheet 2 of 3)

Index	Valid Values	Description
5	0 to 16777215	The fill colour for a target. If this value is negative then it is interpreted as an image index. Line segments and paths ignore this parameter.
6	0 to 16777215	The stroke, or outline colour for a target. Images cannot be used for the stroke. Line segments and paths only use this colour.
7	0 to 1	The width of the stroke or outline to draw on the target. In the case of line segments and paths this is the width of the line. This value is in meters.
8	0 to 499	When a target type is >100 (i.e. a background target) then this is the ID of the target.
9	0 to 100	The is the opacity of the target. 0 means the target is completely transparent, 100 means the target is completely opaque.
10	0 to 10	For circles this is the radius of the circle in meters.
10, 11, 12	0 to 10	For ellipses these are (in order) the size of the major radius, minor radius and the rotation of the target. All measures are in meters, rotation is in radians.
10, 11, 12	0 to 10	For rectangles these are (in order) the width (X direction), height (Y direction) and rotation of the target. All measures are in meters, rotation is in radians.
10, 11	0 to 10	For Line segments this is the offset in X, and Y to the end point of the line. The offsets are in meters,
10, 11, 12, 13	0 to 10	For triangles these are (in order) the width (X direction), height (Y direction), peak offset from midpoint and rotation of the target. All measures are in meters, rotation is in radians.
10	-2*pi to 2*pi	For polygons this is the rotation of the polygon in radians.
14	n/a	Not used
15 to 64	0, 32 to 126	For targets of type 11 to 15 and 111 to 115 these are values for ASCII characters that will be used to display text. ex. a=97
65	0 to 16777215	For targets of type 11 to 15 and 111 to 115 this is the colour of the text
66	0 to 1	For targets of type 11 to 15 and 111 to 115 this is the height of the text in meters.

Table 14-6: VCode Vector Format (Continued) (Sheet 3 of 3)

Index	Valid Values	Description
16	1 to 25	For paths this is a count of the path vertexes. For polygons this is a count of the polygon vertexes. For repeat targets this is a count of the repeat locations.
17 to 66	-10 to 10	For paths these are the path vertex locations in meters in global coordinates. Path targets do not use indexes 3 and 4 in the VCode. For polygons these are the polygon vertex locations in meters relative to the polygon centre (the polygon centre is indexes 3 and 4 in the VCode). For repeat targets these are the locations of the targets in meters in global coordinates. Repeat targets do not use indexes 3 and 4 in the VCode. The coordinates here are stored as [X0, Y0, X1, Y1, X2, Y2].
68 to 70	n/a	Not used

Table 14-7: Vector Indices Used for Different Shape Types

Shape - Value of Element 1	Indices Used
Circle - 1	[1-7, 9-10]
Ellipse - 2	[1-7, 9-12]
Rectangle - 3	[1-7, 9-12]
Line segment - 4	[1-4,6,7, 9-11]
Triangle - 5	[1-7, 9-13]
Path - 6	[1-2, 6-7, 9, 16, 17-66]
Polygon - 7	[1-7, 9, 10, 16, 17-66]
VCodes with text - 11 - 15, for example: 10 + basic shape ID	Indices used for the basic shape from above plus [15-64, 65, 66]
VCode repeats - 21 - 25, for example: 20 + basic shape ID	Indices used for the basic shape from above plus [16, 17-66] but without [3, 4]
Background - various within the range of 101 to 125. For example, 100 + a non-background ID.	Indices used for the non-background definition plus [8]

14.7.7 VCode Examples

Some example VCodes are:

- **Circle example** - [1, 1, 0.0, 0.1, 255, 0, 0, 0, 100, 0.02...] - This VCode will create a 2 cm radius blue circle drawn on the mid-line, 10 cm from the front of the VR.
- **Rectangle example** - [3, 1, 0.2, 0.3, 65280, 128, 0.01, 0, 100, 0.2, 0.1, 0.7854...] - This VCode will create a 20 cm x 10 cm green rectangle with a dark blue 1 cm border. The rectangle is drawn at right of mid-line by 20 cm and 30 cm from the front of the VR and be drawn at a 45 degree angle (0.7854 radians = 45 degrees).
- **Repeat target example** - [21, 1, 0, 0, 4194304, 0, 0, 0, 50, 0.005, 0, 0, 0, 0, 0, 3, -0.1, 0.2, 0.0, 0.2, 0.1, 0.2...] - This VCode will create a repeat target. This will draw 3 dark red circles that are 0.5 cm radius. The circles will be in a line 20 cm from the front of the VR at 10 cm left of mid-line, on mid-line, and 10 cm right of mid-line. The circles are all drawn at 50% transparency.
- **Polygon example** - [107, 1, -0.3, 0.4, 255, 0, 0, 10, 100, 0, 0, 0, 0, 0, 0, 4, 0.01, 0.01, 0.01, -0.01, -0.01, -0.01, -0.01, 0.01...] - This VCode will create a background target with ID 10. The target is a blue polygon with 4 vertexes. The polygon is centred at 30 cm left of mid-line and 40 cm from the front of the VR. The vertex points make a square of length and width 2 cm.

NOTE: Certain VCode errors can be detected by Dexterit-E and reported to the operator. These errors will also be saved with your data file as a system generated Task Event Code (see *Section: 14.13 System Generated Task Events Codes*).

14.8 Data Saving and Logging

Data collected by a Task Program are saved in exam ZIP files. For more information on how to access the data in these exam files, please see the Dexterit-E User's Guide.

Which data are logged by a given Task Program depends on the `DataLogging` block. The available check boxes mean:

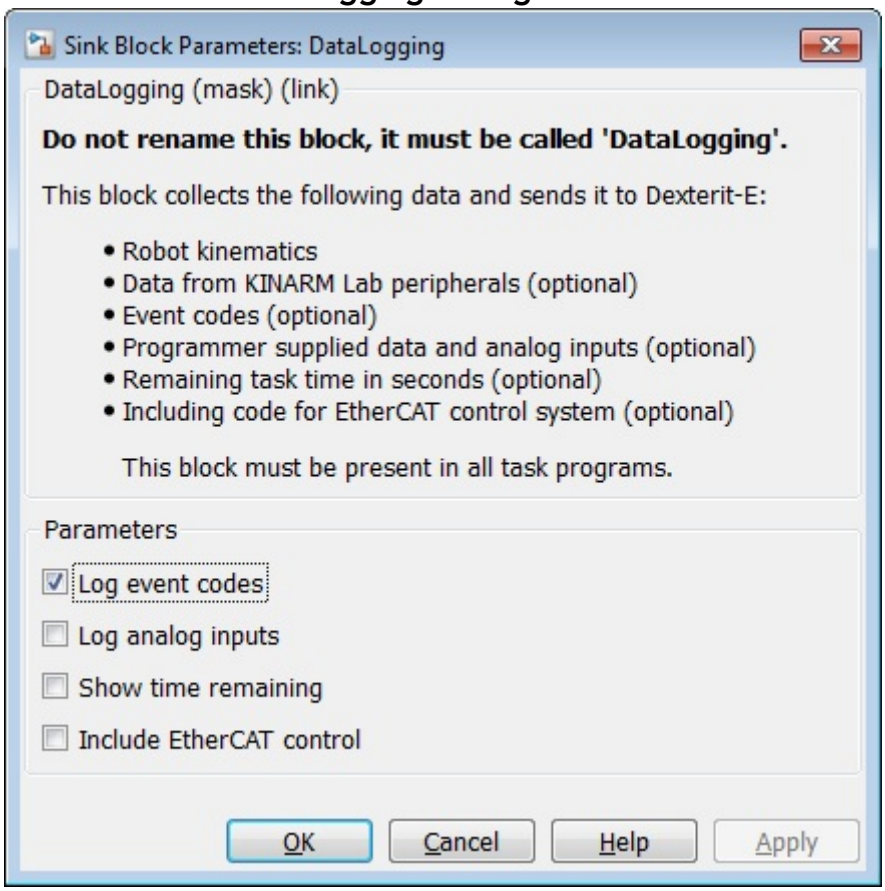
- **Log event codes**. Creates an input called `event_codes`. Numbers sent to the input are used to record Task Event Codes.
- **Log analog inputs**. Creates and input called `analog_data`. This input can be used to record true analog inputs from an analog input card, or it can be used to record custom data that are created on each time step (e.g. calculated positions of moving targets).
- **Show time remaining**. Creates an input called `seconds_remaining` that allows you to input the number of seconds remaining in the task. This value is then used within Dexterit-E to show the operator how much time is left in the task.
- **Include EtherCAT control**. When checked the compiled task will be able to run on any Kinarm Lab. When unchecked the task will only be able to run on Kinarm Classic Labs and Kinarm EP Rev 1 Labs. If you have one of these labs and you do not intend to share your task with other labs, then unchecking this option will reduce compile times within MATLAB. If you compile your task with this option unchecked, and

then you try to run your task on a Kinarm Lab that requires EtherCAT control, Dexterit-E will provide an appropriate error message. This option is checked by default.

– To determine if you have an EtherCAT controller, go to Kinarm Lab® Hardware Configuration and look at the selection for Control type.

Kinarm kinematic data (e.g. positions, velocities, torques, etc.) are recorded automatically.

Figure 14-3: Screenshot of DataLogging Dialog



When data are logged depends on the `logging_enable` input and on the run/pause state of a task. Data are only saved if `logging_enable=1` and if the task is running. Data are not saved when a task is paused.

14.9 Available 'Tags' (From and Goto Blocks)

Section: 10.16 Accessing KINdata_bus, Current Block Index, and Other Task Control Variables demonstrated how to use Simulink's From block to access Goto tags. A list of available tags and what data they contain is provided here.

Table 14-8: Available From Tags (Sheet 1 of 8)

Tag	Description
`Block_Definitions`	Partial copy of the Block Table for the current Task Protocol, albeit in a different format than seen by the operator in Dexterit-E. In this format, the columns of `Block_Definitions` are as follows: 1 – Randomize (0 = non-random, 1 = random); 2 to 500 – list of Trial Protocols. If a block uses bracketed lists to define the TP List (i.e. round or square brackets) and the block is randomized, then the order of the Trial Protocols is randomized by Dexterit-E before being passed into this `Block Definitions` variable, and the first element of the row will be 0 in order to indicate that the TDK should not apply any further randomization. If a block does not use bracketed lists to define the TP List then the order of Trial Protocols in the `Block Definitions` row will be as it was entered in the Block Table, and the first element of the row will be 0 or 1 to indicate if randomization is to be handled in the TDK.
`Block_Sequence`	Partial copy of the Block Table for the current Task Protocol, albeit in a different format than seen by operators in Dexterit-E. In this format, the columns of `Block_Sequence` are as follows: 1 – Block index (i.e. row number of Block Table or `Block_Definitions`) 2 – Number of repeats of block defined by block index.
ECATDigitalDiagnostic	Reserved for internal use.
ECATInitStatus	For EtherCAT controlled systems this is the output of the EtherCAT Init block. Please see the MATLAB documentation for the EtherCAT Init block.
`EL_Camera_Angle`	Reserved for internal use.
`EL_Camera_Focal_Length`	Reserved for internal use.
`EL_Camera_Position`	Reserved for internal use.
`EL_Tracking_Available`	Reserved for internal use.
`ForcePlateData`	Matrix containing data from Kinarm Lab force plates. See *Section: 14.3 Force Plate Data*.

Table 14-8: Available From Tags (Continued) (Sheet 2 of 8)

Tag	Description
Gaze_Global	Matrix containing data from Kinarm Gaze-Tracker. See *Section: 14.4 Kinarm Gaze-Tracker Data*.
Gaze_PupilArea	See *Section: 14.4 Kinarm Gaze-Tracker Data*.
Gaze_Event_Info	See *Section: 14.4 Kinarm Gaze-Tracker Data*.
Gaze_Vector_Global	See *Section: 14.4 Kinarm Gaze-Tracker Data*.
Gaze_Pupil_Area	See *Section: 14.4 Kinarm Gaze-Tracker Data*.
Gaze_Tracking_available	1 indicates that the Kinarm Lab is enabled with a Kinarm Gaze-Tracker, 0 indicates gaze tracking is not available. This does not indicate whether gaze tracking has been calibrated or not.
KINARMEPScalingRobot1	The scaling factor currently being applied to any force on an End-Point robot 1. This tag is only available if the KINARM_EP_Apply_Loads block is present in the model.
KINARMEPScalingRobot2	The scaling factor currently being applied to any force on an End-Point robot 2. This tag is only available if the KINARM_EP_Apply_Loads block is present in the model.
KINARMEPTorqueCMD	A 1x4 vector of current torque command to Kinarm EP robots. Click Help for the KINARM_EP_Apply_Loads block for description of elements. This tag is only available if the KINARM_EP_Apply_Loads block is present in the model.
KINARMScalingRobot1	The scaling factor currently being applied to any torque on Kinarm Exoskeleton robot 1. This tag is only available if the KINARM_Apply_Loads block is present in the model.
KINARMScalingRobot2	The scaling factor currently being applied to any torque on Kinarm Exoskeleton robot 2. This tag is only available if the KINARM_Apply_Loads block is present in the model.
KINARMTorqueCMD	A 1x4 vector of current torque command to Kinarm robots. Click Help for the KINARM_Apply_Loads block for description of elements. This tag is only available if the KINARM_Apply_Loads block is present in the model.

Table 14-8: Available From Tags (Continued) (Sheet 3 of 8)

Tag	Description
KINARM_docking_points	Global position (x, y) of Kinarm Exoskeleton's calibration docking points (m).
KINARM_Lab_info_bus	This is a bus signal containing detailed information about the hardware for the current Kinarm Lab. Information includes: the count of Kinarm Robot arms, the presence of a Kinarm Gaze Tracker, the presence of a robot lift, the type of the robot (Kinarm End-Point, or Kinarm Exoskeleton), the presence of secondary encoders and the presence of force torque sensors.
KINdata	Matrix containing data relevant to current Kinarm Lab. This option has been superceded by the KINdata_bus and will become obsolete in a future version of Dexterit-E.
KINdata_bus	A bus signal containing the same data as KINdata but with named signals for each variable.
Load_Table	Copy of the Load Table.
Run_Status	Status of task.(0 = stopped/paused, 1 = running)
TP_Table	Copy of Trial Protocol table.
TP_row	The currently active row from the Trial Protocol table. This is the same as current_tp_index.
Target_Labels	When targets are defined with text labels the text is stored in this table as one label per row.
Target_Table	Copy of Target Table. When the Target Table is accessed in a Task Program, the positions of targets are in global coordinates, not in the local coordinates viewed in Dexterit-E.
Task_Wide_Parameters	The contents of the Task Wide Parameters table.
current_block_index	Block index of current block. The block index refers to the row in the Block Table in Dexterit-E (i.e., the row of the Block_DefinitionsFrom tag).

Table 14-8: Available From Tags (Continued) (Sheet 4 of 8)

Tag	Description
current_block_number_in_set	Block number of current block. In the Block Table in Dexterit-E, a single row may define multiple instances of identical blocks (i.e. if Block Reps > 1). Each of those repeats counts as a separate block for the purposes of this counter. This number begins at 1 and increments with each new in block throughout the Task. Its final value will equal the total number of blocks defined in the Block Table (i.e. sum of values in the Block Reps column).
current_tp_index	Row number of current Trial Protocol in the TP_Table (i.e. can be used with TP_Table to extract the current trial protocol).
current_trial_number_in_block	Number of current trial in block. This number begins at 1 for each block and increments with each new trial throughout that block. If a Task is paused between blocks, then current_trial = 0 will occur during that pause.
current_trial_number_in_set	Number of current trial in set. This number begins at 1 for each set and increments with each new trial throughout the set.
display_size_m	Size of the subject display (x, y) in meters.
display_size_pels	Size of the subject display (x, y) in pixels.
encoderDeltas	This is a 4 element vector, one for each joint showing the difference between the primary and secondary encoder angles.
e_end_trial	This is the signal sent from the Stateflow diagram to indicate the current trial is over.
e_exit_trial_now	This is the signal sent to the Stateflow diagram to indicate the current trial should be stopped (e.g., Pause was clicked).

Table 14-8: Available From Tags (Continued) (Sheet 5 of 8)

Tag	Description
`exam_progress_bus`	This bus signal controls several values that indicate the current state of an exam: • trial_number_in_exam - the count of trials run, starts at 1 when an exam is started. • total_trials_in_exam - the count of trials to expect in an exam. • trial_number_in_block - the count of trials in the block so far, starts at 1 when a block is started. • block_number_in_exam - the count of blocks run so far, starts at 1 when an exam is started. • total_blocks_in_exam - the count of blocks expected in the exam. • tp_row - the current Trial Protocol row being used. • block_row - the current block row being used.
`global_clock`	Square wave signal at 1 kHz.
`last_frame_acknowledged`	Frame number of the last video frame that was acknowledged as displayed by the Dexterit-E Computer.
`last_frame_sent`	Frame number of the last video frame that was sent to the Dexterit-E Computer.
`primary_encoder_data`	Matrix of robot data determined from the primary encoders. This tag allows access to primary encoder data even when using optional secondary encoders. This matrix has two rows, one for each robot. Each row has six elements: 1 – L1 position (rad). 2 – L2 position (rad). 3 – L1 velocity (rad/s). 4 – L2 velocity (rad/s). 5 – L1 acceleration (rad/s^2). 6 – L2 acceleration (rad/s^2).
`primary_encoder_data_bus`	The same as `primary_encoder_data` but in a bus format.

Table 14-8: Available From Tags (Continued) (Sheet 6 of 8)

Tag	Description
`robot1_calibration_parameters`	Vector of information about the calibration of robot_1. Elements of this vector are: 1 - Shoulder angle offset (rad). 2 - Elbow angle offset (rad). 3 - Shoulder x position (m). 4 – Shoulder y position (m). 5 – L1 length (m). 6 – L2 length (m). 7 – Offset to pointer from L2 (m). 8 – L3 error estimate (m).
`robot1_calibration_parameters_bus`	The same as `robot1_calibration_parameters` but in a bus format.
`robot2_calibration_parameters`	Vector of information about the calibration of robot 2, if present. The format of this vector is the same as `robot1_calibration_parameters`.
`robot2_calibration_parameters_bus`	The same as `robot2_calibration_parameters` but in a bus format.

Table 14-8: Available From Tags (Continued) (Sheet 7 of 8)

Tag	Description
`robot1_motor_parameters`	Bus of information about the hardware of robot 1. • Arm Orientation – 1 = right exoskeleton / left endpoint, -1 = left exoskeleton / right endpoint. • M1 orientation motor 1 orientation (1 = motor shaft down, -1 = motor shaft up). • M2 Orientation motor 2 orientation (1 = motor shaft down, -1 = motor shaft up) • M1 Gear Ratio - motor 1 gear ratio. • M2 Gear Ratio - motor 2 gear ratio. • Has Secondary Enc - using secondary encoders (1 = using secondary encoders, 0 otherwise). • Robot type (1 = endpoint, 0 = exoskeleton, 2=NHP). • Robot version - 1 = classic, 2=New. • Torque constant - Conversion factor for reading torque from DAQ. (Torque constant / 10v) * DAQ voltage = applied torque. • isEP - is 1 if the robot is an End Point. • isHumanExo - 1 if the robot is a human Kinarm Exoskeleton. • isClassicExo - 1 if the robot is the original (pre-2016) design of the Kinarm Exoskeleton, now referred to as the Kinarm Classic. • isUTSExo - 1 if the robot is the new (2016) design of the Kinarm Exoskeleton. • isPMAC - 1 if the robot is controlled by a PMAC. • isECAT - 1 if the robot is controlled by EtherCAT. • robotRevision - the revision number of the robot.
`robot2_motor_parameters`	Bus of information about the hardware of robot 2, if present. The format of this vector is the same as `robot1_motor_parameters`.

Table 14-8: Available From Tags (Continued) (Sheet 8 of 8)

Tag	Description
`run_state`	State of Task Program running on the Robot Computer, with the following definitions: 1 – Task is ready but has not yet started. 2 – Task is running. 3 – Task is paused. 4 – Task is finished.
`seed`	A number used to seed the random number generator. Depending on the settings of the Task Protocol, this may be fixed or it may be set by Dexterit-E at run-time.
`subject_height`	Subject height of the active subject as entered into the Dexterit-E user interface (m).
`subject_weight`	Subject weight of the active subject as entered into the Dexterit-E user interface (kg).
`target_frame_of_reference_center`	The (X,Y) position of the frame of reference centre used to calculate target positions (m).
`task_control_button`	Button number (as defined in the `Parameter Table Defn` block) of the currently clicked Task Control Button. The value will be equal to the button number for 1 ms when the Task Control Button is clicked by the operator in the Dexterit-E GUI. The value will revert back to zero on the following time-step.

14.10 GUI Control Block Input Events and Output Events

- **e_clk** – This input event must be on port 1 and must have a defined trigger of Rising. This event occurs every 1.0 ms, as long as the Task Program is running (i.e. as long as the task has started and is not paused). This event causes every Stateflow Chart to which it is wired to execute once per ms (because Stateflow Charts only execute when an event occurs). A typical secondary usage of `e_clk` is as a time counter (e.g. see usage of `after` in *Figure: 6-11*)
- **e_ExitTrialNow** – This input event must be the on port 2 and must have a defined trigger of Either. This event occurs whenever an operator clicks Pause and Dexterit-E has been set to pause the trial immediately. This event can be used by

the end-user to define what will happen in the Task Program when this event occurs. For an example, see *Section: 6.13 Including Pause Button Functionality*.

- **e_Trial_End** – This output event can be on any port, but must have a defined trigger of Either. This event is used by the end-user to signal when a trial is over so that the Task Program can update itself for the next trial (e.g. see usage in `Between_Trials` state in *Figure: 6-11*).

14.11 Task Protocol Parameter Table Sizes

Each of the tables for a Task Protocol has a maximum size that puts an upper limit on the number of parameters that can be specified. Some of the table dimensions are fixed, while others have default values that can be specified by the task programmer when they build the task. The possible sizes for each table are:

- **Target Table** - 10-2000 targets; defaults to 64; 2-25 parameters each.
- **Load Table** - 10-2000 loads; defaults to 20; 0-20 parameters each.
- **Trial Protocol Table** - 10-2000 trial protocols; defaults to 100; 0-50 parameters each.
- **Task Wide Parameters Table** - 0-50 parameters.
- **Block Table** - 50 blocks.
- **Analog Inputs Table** - 0-50 analog inputs.

If you require more rows in the Target, Load, or TP tables, you can double-click on the `GUI Control` block and navigate to the **Tables** tab. Each table must have a minimum of 10 rows and a maximum of 2000 rows.

The number of parameters that can be specified in each of the tables is set using the `Parameter Table Defn` block. See *Section: 6.12 Using the Parameter Table Defn Block*.

14.12 Reserved IP Addresses

It is possible to add additional hardware or computers to the Dexterit-E LAN to communicate with the Robot Computer. However, there are limitations on the IP addresses that can be assigned. The IP address must be of the form 192.168.0.x, where x is not reserved or already used by a Kinarm Lab. Table below contains a map of the used, reserved and available IP addresses.

Table 14-9: Reserved IP Addresses (Sheet 1 of 2)

IP Address	Function
192.168.0.1	Dexterit-E Computer
192.168.0.2	Robot Computer
192.168.0.3	Kinarm Gaze-Tracker Computer

Table 14-9: Reserved IP Addresses (Continued) (Sheet 2 of 2)

IP Address	Function
192.168.0.4-9	<Reserved>
192.168.0.10-11	Kinarm EP Force/Torque Sensors
192.168.0.12-20	<Reserved>
192.168.0.21-99	<Available for custom-defined use>
192.168.0.101-102	Kinarm Force Plates
192.168.0.102-109	<Reserved>
192.168.0.110-255	<Available for custom-defined use>

14.13 System Generated Task Events Codes

As part of the normal operation of a Kinarm Lab there are several Task Event Codes that may be generated during the running of a task and which are saved to the data file along with user-defined Task Event Codes as "events". The system generated Task Event Codes have reserved labels that cannot be used when creating your own Task Event Code labels in the `Parameter Table Defn` block (*Section: 10.1 Task Event Codes*).

Table 14-10: System Generated Task Event Code Labels (Sheet 1 of 2)

Task Event Code Label	Meaning
E_NO_EVENT	This is a system defined event that is not recorded. This event code can be used in Simulink in order to reset the state of the `event_code(s)` signal.
CHAIR E-STOP [PRESSED | RELEASED]	The operator pressed or released the emergency stop button that is part of the chair on a Kinarm Exoskeleton Lab. This message is not available for Kinarm Classic Labs.
DESKTOP E-STOP [PRESSED | RELEASED]	The operator pressed or released the desktop (i.e. hand-held) emergency stop button. This message is not available for Kinarm Classic Labs.
TASK_PAUSED	The operator clicked the **Pause** button in Dexterit-E.
TASK_RESET	The operator clicked the **Reset** button in Dexterit-E.

Table 14-10: System Generated Task Event Code Labels (Continued) (Sheet 2 of 2)

Task Event Code Label	Meaning
GAZE BLINK START	The Kinarm Gaze-Tracker determined that the subject started a blink.
GAZE BLINK END	The Kinarm Gaze-Tracker determined that the subject ended a blink.
GAZE SACCADE START	The Kinarm Gaze-Tracker determined that the subject started a saccade.
GAZE SACCADE END	The Kinarm Gaze-Tracker determined that the subject ended a saccade.
GAZE FIXATION START	The Kinarm Gaze-Tracker determined that the subject started a fixation.
GAZE FIXATION END	The Kinarm Gaze-Tracker determined that the subject ended a fixation.
GAZE FIXATION UPDATE START	The Kinarm Gaze-Tracker determined that the subject started to change his or her fixation location.
GAZE FIXATION UPDATE	The Kinarm Gaze-Tracker determined that the subject ended changing his or her fixation location.
[RIGHT | LEFT] HANDLE [GRASPED | RELEASED]	The subject has grasped or released the handle of a Kinarm End-Point Lab. This event is not available for Kinarm End-Point Classic Lab.
TASK_PAUSED	The operator clicked the **Pause** button in Dexterit-E.
TASK_RESET	The operator clicked the **Reset** button in Dexterit-E.
VCODE_INVALID_<various>	The TDK has detected an error in a VCode that was sent to the `Process_Video_CMD`block. This event code indicates that there is a programming bug in the Simulink model file that needs to be addressed by the end-user.
VCODE_TOO_MANY_TARGETS	
VCODE_ERROR_[id]	

14.14 Recording Limits

When recording an exam there are several different limits that should be kept in mind to ensure data integrity.

- In theory Dexterit-E can record trials of any length, but we only test and support Dexterit-E for trials of up to 30 minutes duration.
- There is a limit of 499 trials per block, and 10,000 block repetitions.
- A single exam file cannot be larger than 4GB.

14.15 Updating Task Programs from Prior Versions of the TDK

Differences between TDK versions are documented in the Dexterit-E release notes. This section explains the changes required in an existing Task Program when updating to a later TDK version.

14.15.1 Updating Task Programs from pre-TDK 3.0

Please refer to older versions of this guide.

14.15.2 Updating Task Programs from Dexterit-E 3.0 to 3.1

In Dexterit-E a new `Analog Inputs` block was added. This block is a drop in replacement for the MATLAB National Instruments PCI-6071E and PCI-6229 analog input cards. The `Analog Inputs` block automatically detects, at run time, which card your Kinarm lab is using. This allows task to be compiled once and run on systems with either type of analog input card. It is highly recommended that you update to using the `Analog Inputs` block so that your code is more portable.

14.15.3 Updating Task Programs from Dexterit-E 3.3 to 3.4

In Dexterit-E the new `Parameter Table Defn` is used to replace the old `cfg.xml` file. Please see section *Section: 6.12 Using the Parameter Table Defn Block* for details on using this block. In order to convert your task to using the `Parameter Table Defn` block you will need to create new entries in the block for everything currently in your `cfg.xml` file (with the exception of Default Task Instructions):

1) Columns in your Target Table;
2) Columns in your Load Table;
3) Columns in your TP Table;
4) Task Event Codes;
5) Task Control Buttons.

You can optionally convert some of your existing table columns to Task Wide Parameters. Once you have successfully added all of the entries in the `Parameter Table Defn` block you will need to carefully examine your Stateflow code to ensure you remove your old defined constant names and only make use of the constants you defined by name in the `Parameter Table Defn` block. A good strategy for managing migrating constant names from your old definitions to the new ones is to delete your constants one at a time, attempt compiling, see what breaks and update code to the new constant names from the `Parameter Table Defn` block.

14.15.4 Updating Task Programs from Dexterit-E 3.4 to 3.5

Dexterit-E 3.5 now provides a method to define default task instructions as part of the `Parameter Table Defn` block, rather than requiring the use of the `cfg.xml` block.

NOTE: Support for the cfg.xml file was discontinued in Dexterit-E 3.6.

14.15.5 Updating Task Programs from Dexterit-E 3.5 to 3.6

As of Dexterit-E 3.6 the cfg.xml file is no longer supported, you must use the `Parameter Table Defn` block to define the parameter table in the Task Protocol.

If you are using MATLAB R2015a SP1 you may find that you get errors compiling related to the need to insert `Rate Transition` blocks. R2015a SP1 is the first version where we require multi-core support from SLRT on the Robot Computer. The requirement to use multiple cores means that Simulink now cannot infer rate transitions all the time and some explicit rate transitions will be required. Review the compile error(s) and they should point you to the places that require a rate transition block.

For users of MATLAB R2015a SP1 it should be noted that compile times have increased significantly. This is a result of the increased complexity of the library and the Windows SDK compiler being quite slow. To reduce compile times you can disable compile time optimizations (**Model Configuration Parameters** -> **Code Generation** -> **Compiler Optimization Level**.

The `KINARM_Apply_loads` block has been renamed in the TDK to be `KINARM_EXO_Apply_Loads`. For most users there will be no problems and this change will automatically be corrected. If your apply loads block displays as a broken link you will need to delete your old copy and import the new version of the block.

This version of the TDK introduces several new buses in the `From` tags section. We highly recommend using these new buses. The original non-bus versions will possibly be removed in a future release.

In the TDK there are two `From` tag values that have changed in an incompatible way since Dexterit-E 3.5. The tags are `robot1_motor_parameters` and `robot2_motor_parameters`. These signals were changed from arrays to bus signals. This change means that if you have made use of these from tags then when you migrate your task to Dexterit-E 3.6 you will need to use a Bus Selector block in order to obtain the parts of the signals your code needs to use. Under most circumstances you would only use these from tags in order to help with supporting more than one Kinarm Lab type.

Many of the commonly used `From` tags have had "bus" variants added. For example there is now a `KINdata_bus From` tag. You are encouraged to make use of the bus variants over the original from tags. Bus signals allow the naming of the variables in the signal - which is far less fragile than indexing into an array.

14.15.6 Updating Task Programs from Dexterit-E 3.6 to 3.7 or 3.8

There are no changes required to the Simulink model for the migration of a task from Dexterit-E 3.6 to 3.7 or 3.8.

14.15.7 Updating Task Programs from Dexterit-E 3.8 to 3.9

In general there are no special changes required to update a task program that worked with Dexterit-E 3.8 to work with Dexterit-E 3.9. The exception is if your task makes use of the Set_Target_In_Background block or your task otherwise makes use of background targets. Any task that makes use of background targets will continue to work with Dexterit-E 3.9. However, the feature is considered deprecated and will be removed in a future release.

If your task makes use of background targets then the task can be updated to send all required VCodes all the time. Since Dexterit-E 3.7 it has been possible to send up to 200 VCodes at a time.When updating your task to send all VCodes all of the time, the following may be helpful information:

- If you encounter CPU Overload errors trying to generate and send all required VCodes then you may need to set the Sample time for any embedded MATLAB blocks that are used to create the VCodes. VCodes are only transmitted at 1 kHz, so changing the Sample time to 0.001 s will have no effect at all on visual stimuli.
- If you require more than 200 VCodes at a time then you may be able to make use of *Section: 14.7.4 Repeat Targets* to specify up to 25 targets with a single VCode.

If your task cannot easily be programmed without the use of background targets then contact **Kinarm Support**.

14.16 Updating Tasks to MATLAB R2019b

Users who have existing Kinarm tasks which were created in versions of MATLAB before R2019b should be aware of an important change to Stateflow behaviour when upgrading.

The default behaviour of `State` blocks in Stateflow has changed. If you compile your existing task without modifying it, then it should work properly.

Code that you have written in a State can execute on one of 3 conditions.

- `Entry` - Only when the State is entered does the code execute
- `During` - For every time step that the State is active the code will execute
- `Exit` - Only when moving to a new state does the code execute

In previous versions of MATLAB if you did not specify one of the above conditions then the code in a State was assumed to execute on `entry` only. In MATLAB R2019b the new default if there is no condition is `during`. Please see the Stateflow documentation for more details.

As you are modifying your older tasks, be sure to add the `entry` condition to states if you would like the same behaviour you have always had. This is especially important if you are using a random number generator in the State. Random numbers that are constantly regenerated will often produce subtly incorrect behaviour (ex. lower than expected average pause times).

14.16.1 I/O Blocks

MATLAB has dropped support for all digital and analog I/O boards sold with Kinarm robots. As such, if you make use of I/O blocks in your model you will need to update your model to make use of the versions of these blocks that are in the Kinarm IO library in the TDK.

When opening your model in R2019b, any affected blocks will appear with the text "Obsolete Simulink Real-Time block". Once the block is replaced with a version from the TDK you will need to reconfigure the settings of the new block to match the old one.

14.17 Update the Stateflow Action Language to MATLAB

Any code written in a Stateflow chart must be written in one of two possible language syntaxes: C or MATLAB. This is referred to as the Action Language for the Stateflow chart. Previous users of the Task Development Kit will be used to using C as the Action Language in their Stateflow. As of Dexterit-E 3.9 most code examples have changed to use MATLAB syntax in Stateflow. This change allows more complex code to be written in a Stateflow chart and means that only one programming language is required for an entire model: MATLAB. Any existing Stateflow chart can be changed to use MATLAB as the Action Language.If you have an existing Simulink model that uses a Stateflow chart with the C Action language, the Action Language can be changed to MATLAB using the following steps.

NOTE: Before updating a Stateflow chart it is highly recommended that you make a back-up of your model.

1. Open the Simulink model for the task that you wish to update.
2. Open the Stateflow chart that you wish to update.

   Without the Stateflow chart open the following automatic conversion process will not happen.

   Each Stateflow chart will need to be opened and updated independently.

3. Open the Model Explorer.
4. In the Model Explorer select the Stateflow chart.
5. In the *General* tab on the right hand side of the Model Explorer, change the Action Language to MATLAB.
6. Click **Apply**

   A message will appear at the top of the Stateflow chart, indicating that the Action Language has changed, with an option to automatically convert the syntax.
7. Click the link to automatically convert the chart syntax.

   All of the C syntax code will be converted to MATLAB syntax.
8. Repeat the above steps for any other Stateflow Charts in the model that you wish to update.
9. Build the model and review for compile errors

   Refer to MathWorks' online documentation on "Differences Between MATLAB and C as Action Language Syntax" for possible issues resulting from the conversion process.