Developing Embedded Targets using Real-Time Workshop Embedded Coder

Workshop featuring Freescale S12x

March 2010

Who Should Attend

This seminar is intended for:

- Embedded software developers
- Control design engineers
- Systems engineers

Prerequisites:

- Knowledge of MATLAB and Simulink
- Familiar with concepts of automatic code generation
- Familiar with concepts of embedded software development

Agenda

I. Introduction (60 minutes)
II. Creating a Host Target (60 minutes)
III. Lunch
IV. Creating an Embedded Target (60 minutes)
V. Verifying an Embedded Target using PIL test (30 minutes)
VI. Conclusion and Next Steps

Detailed Agenda

I. Introduction (60 minutes)
- Simulink code generation
- Class Exercise #1
- Target integration options
  - Algorithm Export
  - Real-Time Target Support Examples
  - Add-on Link and Target Products
  - Build your own link or target (Workshop Focus)
- S12x Example Target

II. Creating a Host Target (60 minutes)
- Create baseline target
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Customize Main file
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
  - Class Exercise #2
- Create blockset
  - Options for integrating legacy code
  - Creating blocks and libraries
  - Class Exercise #3

III. Lunch

Detailed Agenda (Cont.)

IV. Creating an Embedded Target (60 minutes)
- Create baseline target
  - Introduction to Target Language Compiler (TLC)
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Generate Main file for single and multiple rate models
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- Create blockset
  - Options for integrating legacy code
  - Creating blocks and libraries
  - Class Exercise #4

V. Verifying an Embedded Target using PIL test (30 minutes)
- Introduction
- Create PIL Application
- Example PIL

VI. Conclusion and Next Steps

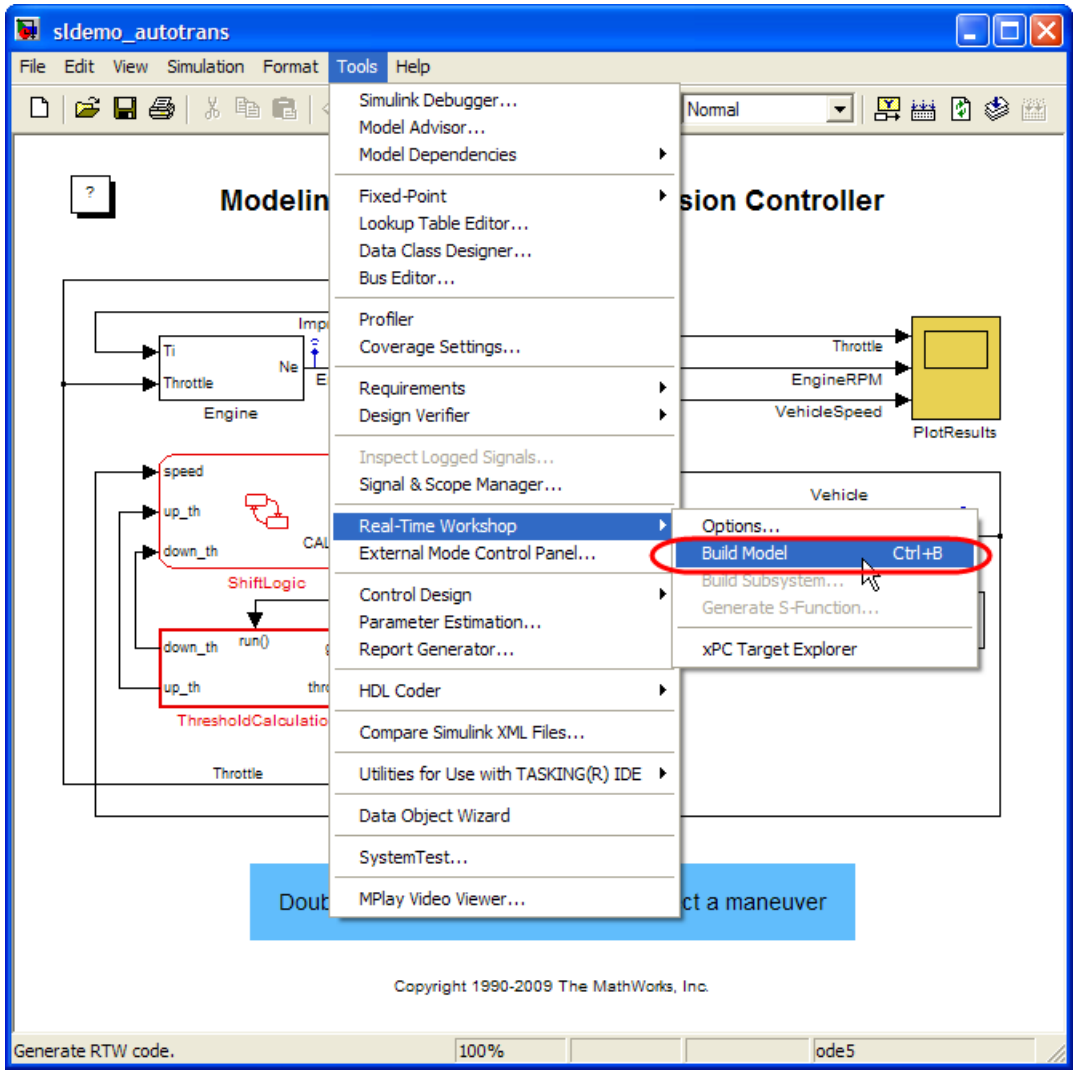
Agenda

I. **Introduction (60 minutes)**
II. Creating a Host Target (60 minutes)
III. Lunch
IV. Creating an Embedded Target (60 minutes)
V. Verifying an Embedded Target using PIL test (30 minutes)
VI. Conclusion and Next Steps

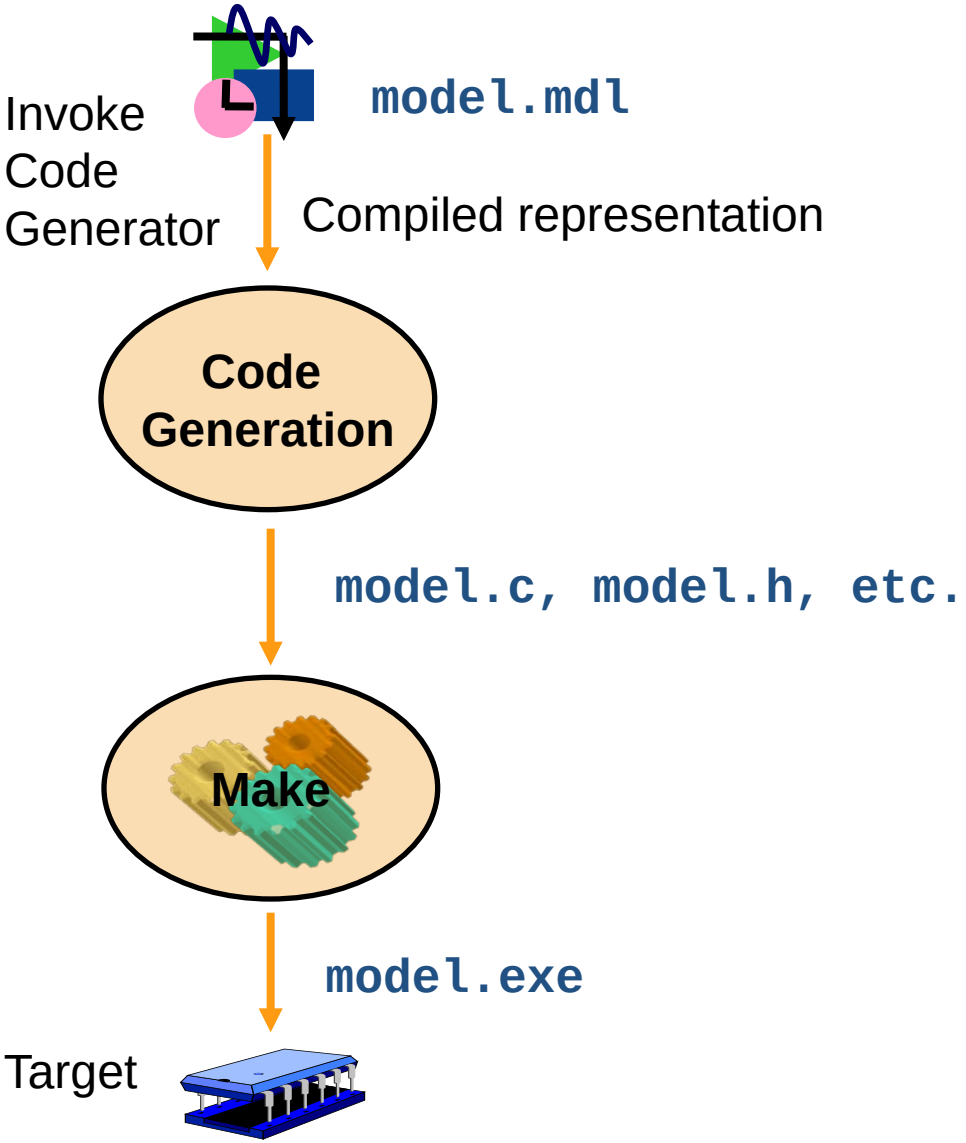
Introduction

- **Simulink code generation**
- Class Exercise #1
- Target integration options
- S12x Example Target

Simulink Code Generation Invocation



Simulink Code Generation Process



Generated Code Files

File: `model.c (or .cpp)`
Contents:
- Variable definitions (nonconstant data)
- Function definitions of
  - `model_initialize()`
  - `model_step()`
  - `model_terminate()`

File: `model.h`
Contents:
- Structure type definitions (model data structures)
- Public interface to
  - Exported data
  - Functions

File: `model_types.h`
Contents:
- Chart structure type definitions (for Stateflow charts and Embedded MATLAB functions)

File: `model_private.h`
Contents:
- Private macros
- External data declarations (imported data)
- External function prototypes

File: `rtwtypes.h`
Contents:
- Defines data types, macros and structure

A customizable main and makefile are also generated and discussed later
A shared utility folder is generated with conditionally generated shared utility files
A model_data.c and rtdata.c files are also conditionally generated

Compiling and Linking

- The generated source code can be either:
  - Compiled on the host platform for host execution
  - Left uncompiled
  - Cross-compiled on the host platform for target execution

Simulink Code Generation Options

Real-Time Workshop

- Generates code from Simulink software that is easy to interact and experiment with
- Uses GRT system target file

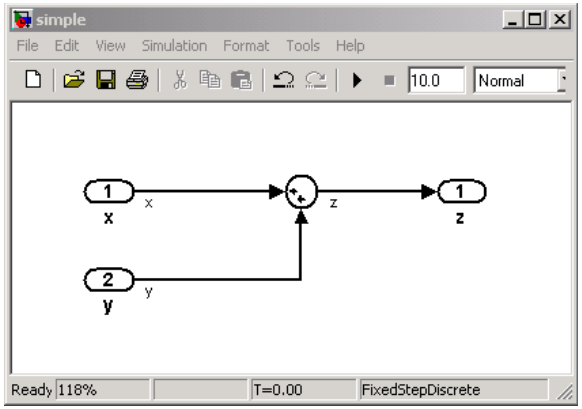
Real-Time Workshop Embedded Coder

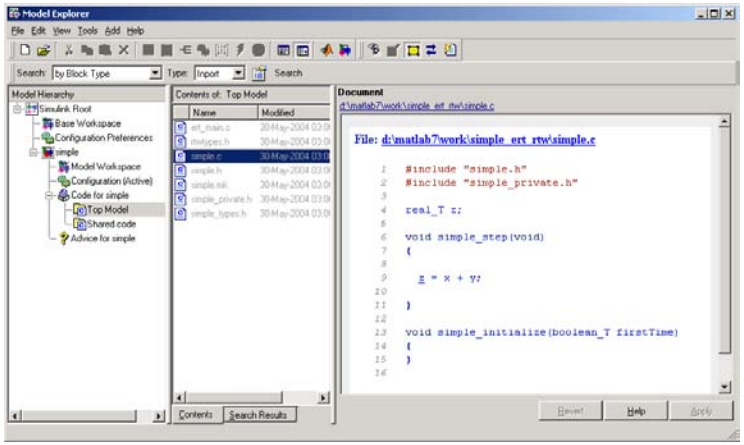
- Generates extremely efficient code that can be customized to look like embedded hand code
- Is an add-on product to Real-Time Workshop
- Uses ERT system target file

You can deploy code on any microprocessor using Real-Time Workshop and Real-Time Workshop Embedded Coder software because they generate standard C/C++ (ANSI/ISO C/C++).

Example

- Generate code from simple model using GRT and ERT
- Examine code generation options





Note that:

- ***Examples*** *are done by the teaching instructor*
- ***Exercises*** *are done by the students*

Class Exercise #1

1. Create a simple model that multiplies two inports and produces one outport
2. Set *lcc* as the compiler by executing “mex –setup” at the MATLAB command prompt
3. Generate code using GRT (default)
4. Generate code using ERT
5. Note the difference in the generated code

6. Time permitting – Experiment with ERT options such as code commenting, bidirectional traceability, and optimizations.

Introduction

- Simulink code generation
- Class Exercise #1
- **Target integration options**
- S12x Example Target

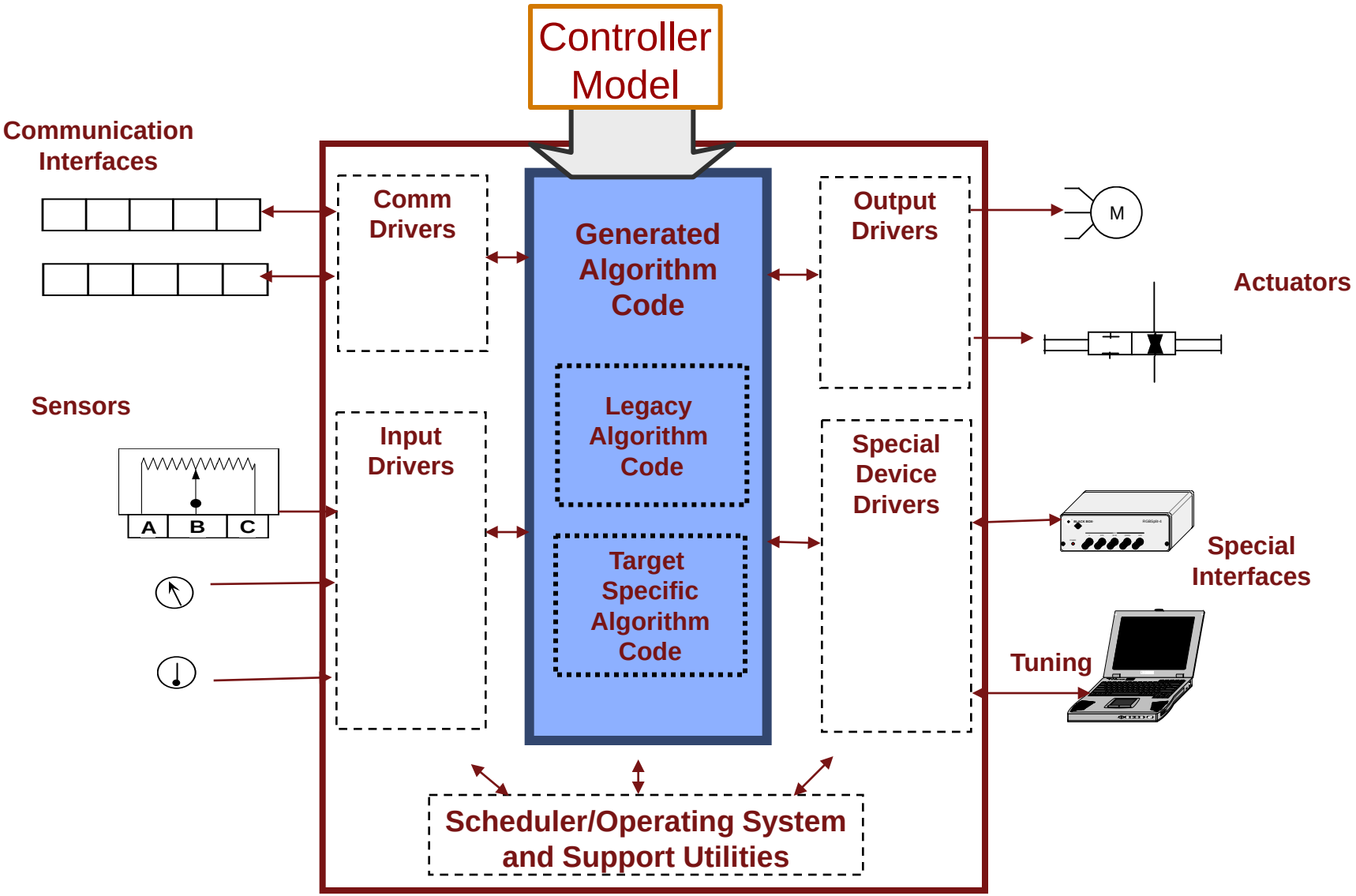
Target Integration Options

- **Algorithm Export**
  - Most popular option for production organizations
- Real-Time Target Support Examples
  - Provided with Real-Time Workshop Embedded Coder
- Add-on Link and Target Products
  - From MathWorks
  - From third parties
- Build your own link or target (Workshop Focus)
  - Using documented procedure and APIs

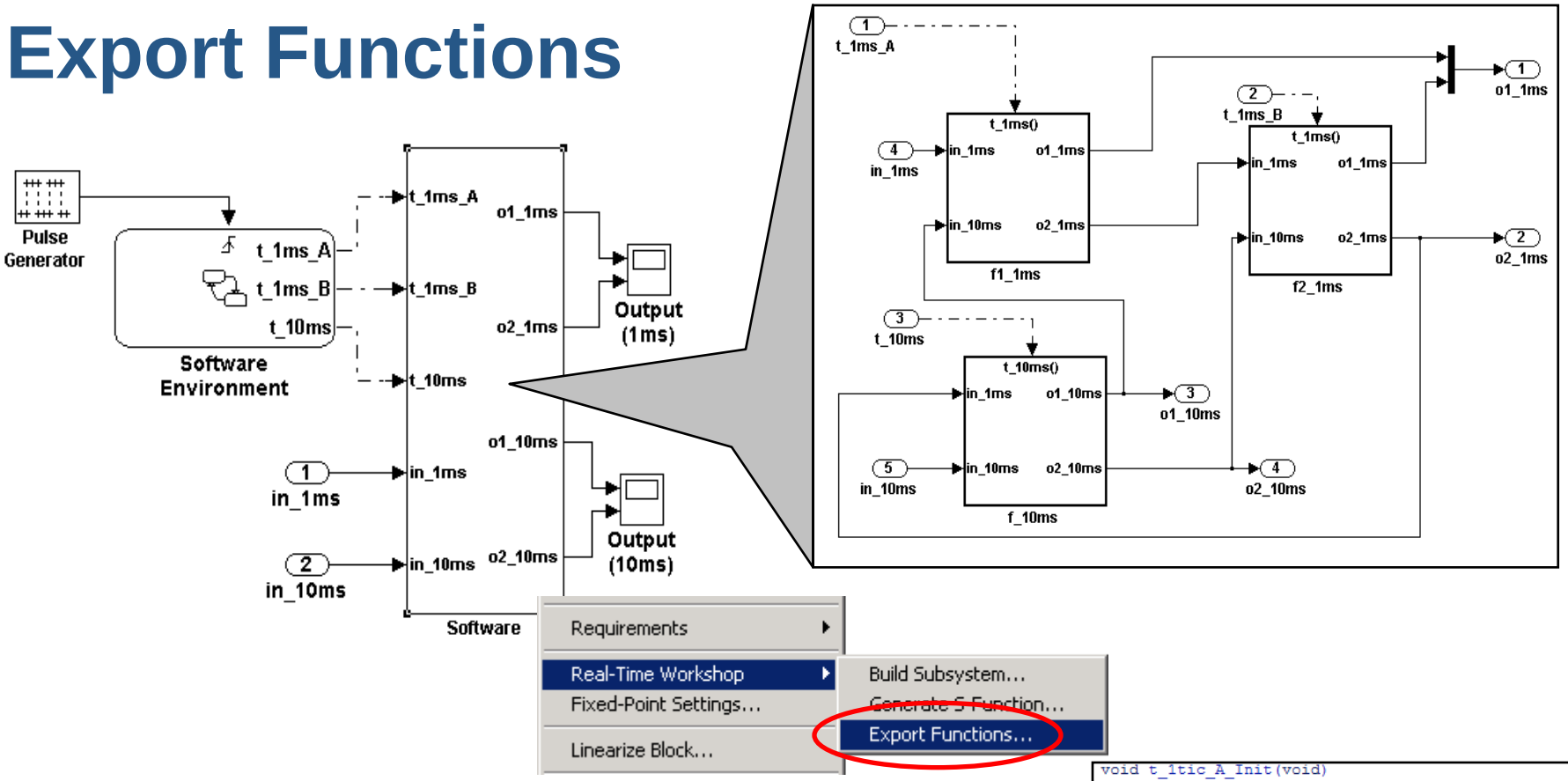
Algorithm Export

- Real-Time Workshop Embedded Coder generates standalone functions callable from external code
- Key Technologies
  - Export functions
  - Model function prototype control
  - Reusable models
  - Legacy Code Tool
  - Target Function Libraries
  - PacknGo (Buildinfo API)
- Third party IDE or make environment
  - Controls the compile and build process
  - Calls generated algorithm code at appropriate time, protects data
  - Integrates RTOS and device drivers

Code Integration – Algorithm Export



Export Functions



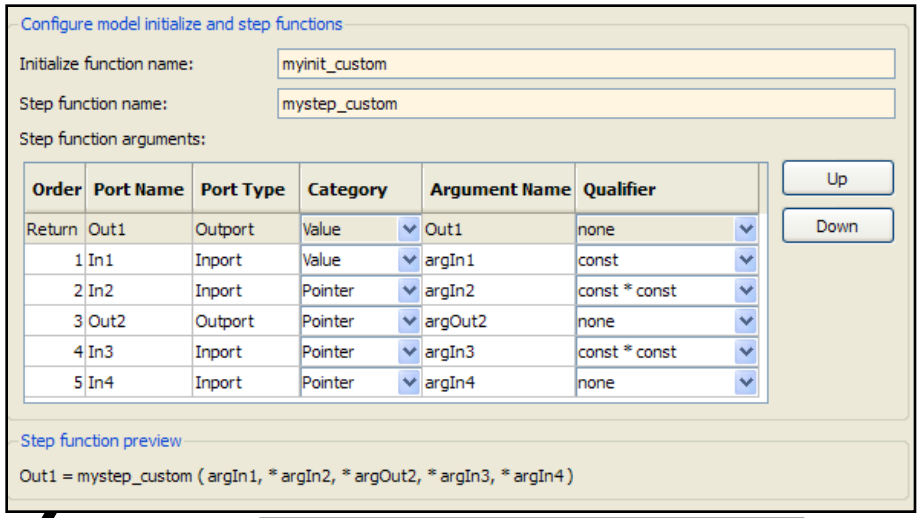
- Supports function-call subsystems
  - A popular scheduling technique in production
- Streamlines code generated
  - No scheduler, no model step function

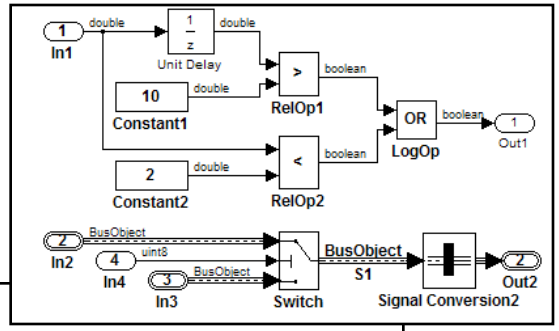
```
>> rtwdemo_export_functions
```

```
void t_1tic_A_Init(void)
{
  rtDWork.Delay_DSTATE_o = 1;
}

void t_1tic_A(void)
{
  rtB.Delay_b = rtDWork.Delay_DSTATE_o;
  rtB.Add = rtU.U1 + (real_T)rtB.Delay;
  rtDWork.Delay_DSTATE_o = (int8_T)(-rtB.Delay_b);
  rtY.TicToc1[0] = rtB.Delay_b;
  rtY.TicToc1[1] = rtB.Delay_c;
}
```

Model Function Prototype Control

- Control model entry functions
  - Pass signals as arguments
  - Pass by value or reference
  - Control names and order
  - Supports Model Reference
- Useful for algorithm export





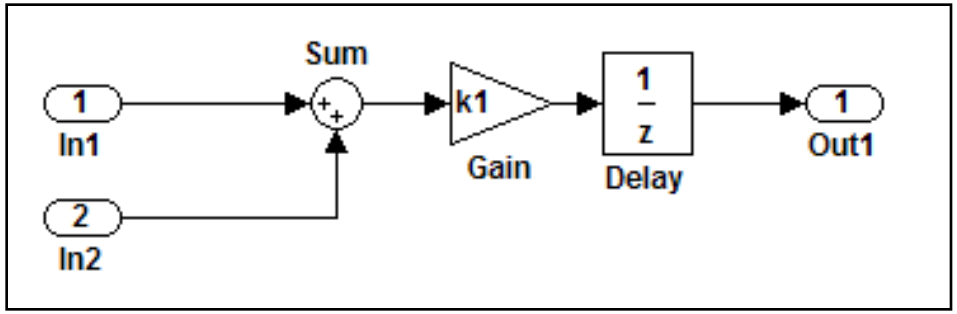

```
/* Model step function */
boolean_T mystep_custom(const real_T argIn1, const BusObject *const argIn2,
  BusObject *argOut2, const BusObject *const argIn3, uint8_T *argIn4)
```

Generated step function definition

```
>> rtwdemo_fcnprotoctrl
```

Reusable Models

- Configures model entry interfaces for reuse
- Passes root I/O and states via arguments



Code interface

GRT compatible call interface

Single output/update function

Terminate function required

Generate reusable code

Reusable code error diagnostic: Error

Pass root-level I/O as: Structure reference

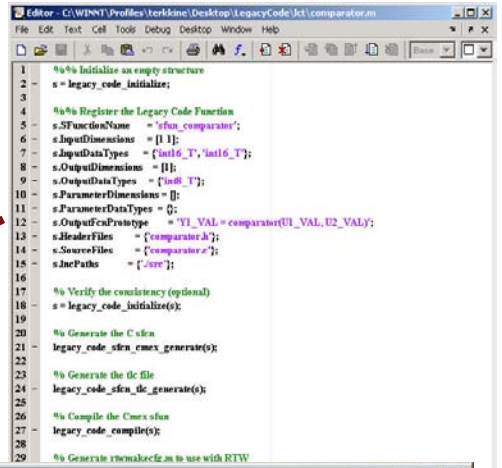
```
/* Model step function */
void rtwdemo_reusable_step(Parameters *rtP, D_Work *rtDWork, ExternalInputs *rtU,
  ExternalOutputs *rtY)
```

```
>> rtwdemo_reusable
```

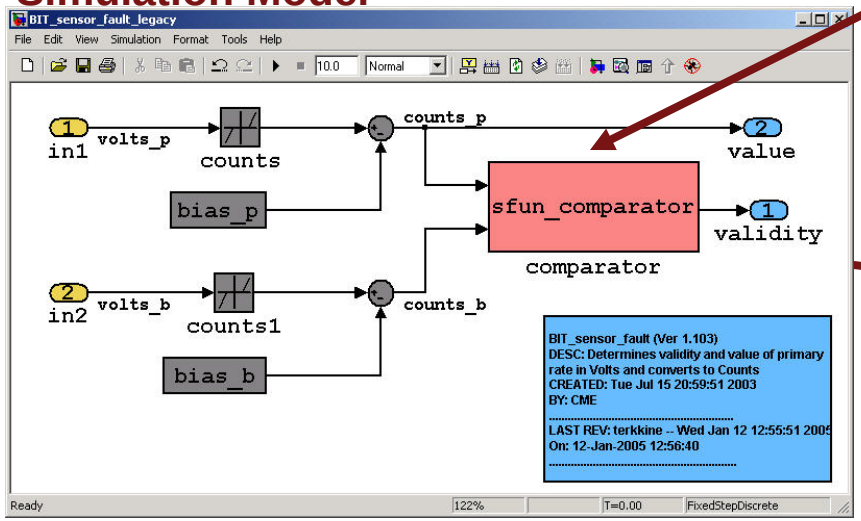
Legacy Code Tool

- Integrates external code
  - For simulation and code generation
  - Demos include lookup tables, filters, etc.

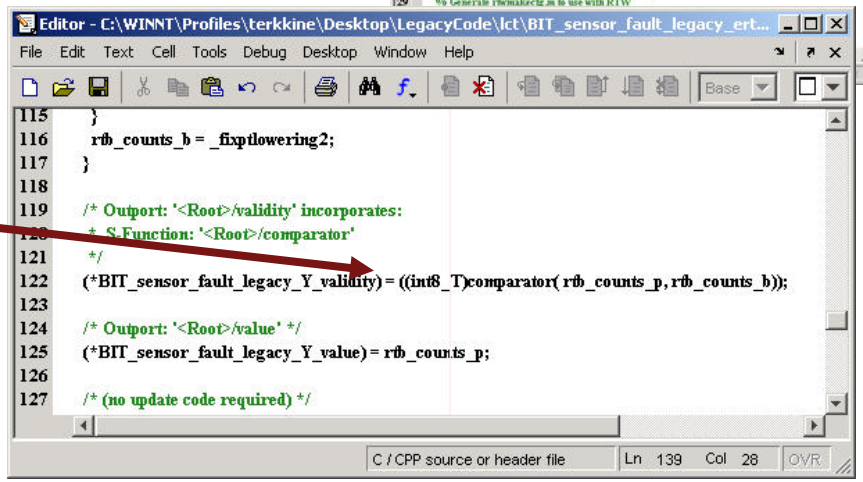
Registration File



Simulation Model



Generated Code




```
>> rtwdemo_lct_filter
```

Target Function Libraries

Problem

- ANSI-C code can be further optimized for processors with built-in arithmetic intrinsic instructions.

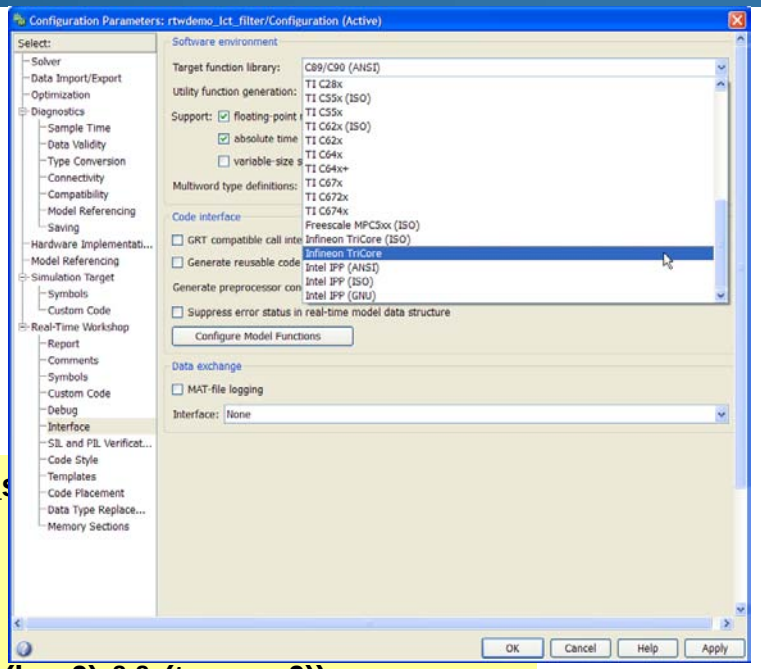
Solution

- Target Function Library Replacement API generates optimized, processor-specific arithmetic code.
- Supported by Embedded MATLAB, Simulink, Stateflow, links, and targets

Benefits

- Reduced code size
- Improved execution performance
- Applies generically across all products
- Alternative to generic ANSI C code
- Model- and release-independent

```
>> rtwdemo_tfl
```



ANSI-C Code

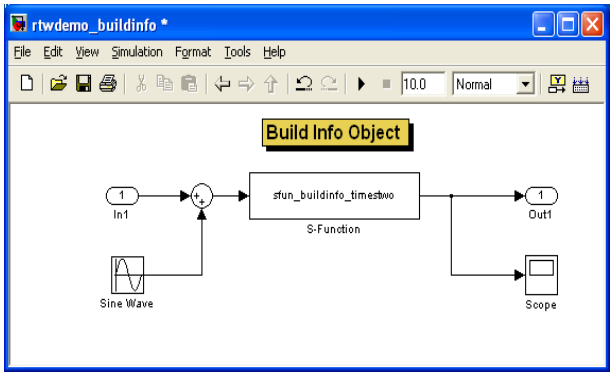
```
int32_T add_s32_
{
        int32_T tmp;

        tmp = a + b;
        if ((a < 0) && (b < 0) && (tmp >= 0))
       {
              tmp = MIN_int32_T;
        }  else if ((a > 0) && (b > 0) && (tmp <= 0)) {
              tmp = MAX_int32_T;
         }
         return tmp;
}
```

Optimized for TriCore

```
int32_T tricore_add_s32_s32_s32_sat(int32_T a,
                                    int32_T b)
{
    return (__sat int)a + b;
}
```

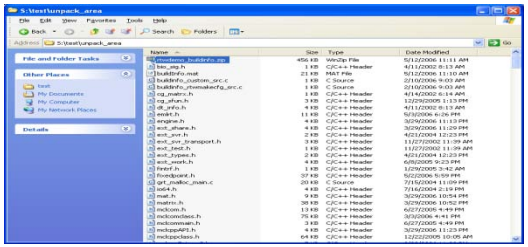
PacknGo for Relocating Generated Code

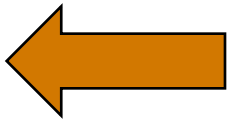
- Lets you rebuilds generated code on any computer
  - Even one without MATLAB
- Creates zip file with generated code and static dependencies
- Based on the BuildInfo API
  - set_param(bdroot, 'PostCodeGenCommand', 'packNGo(buildInfo);');



Code, Pack (Computer A)

Go





Unpack, Build (Computer B)

```
>> rtwdemo_slexprfold
```

Target Integration Options

- Algorithm Export
  - Most popular option for production organizations
- **Real-Time Target Support Examples**
  - Provided with Real-Time Workshop Embedded Coder
- Add-on Link and Target Products
  - From MathWorks
  - From third parties
- Build your own link or target (Workshop Focus)
  - Using documented procedure and APIs

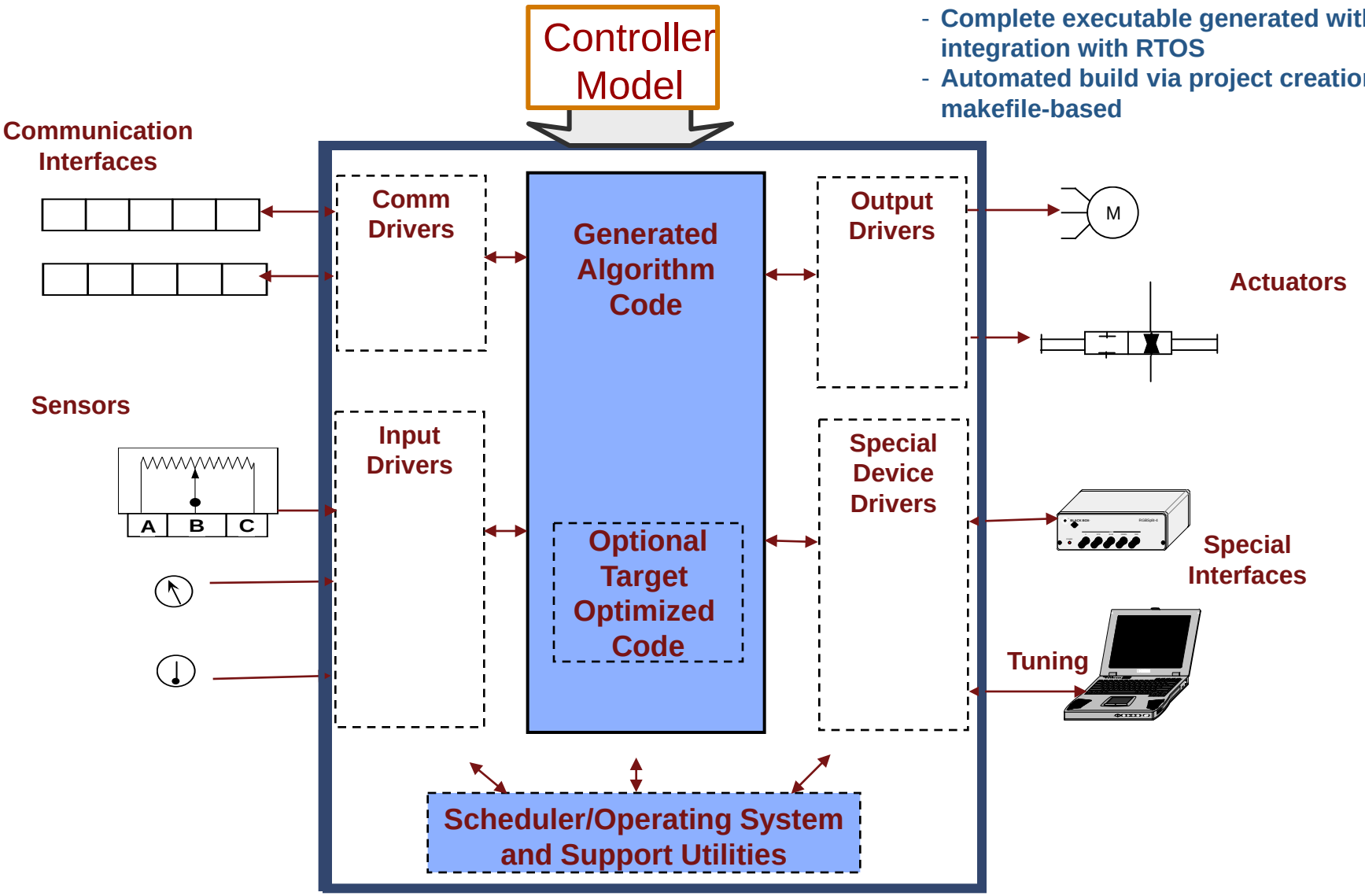
Real-Time Target Support Examples

- Real-Time Workshop Embedded Coder provides:
  - VxWorks Target
  - OSEK Target
  - AUTOSAR Target

Production Code Generation and Integration

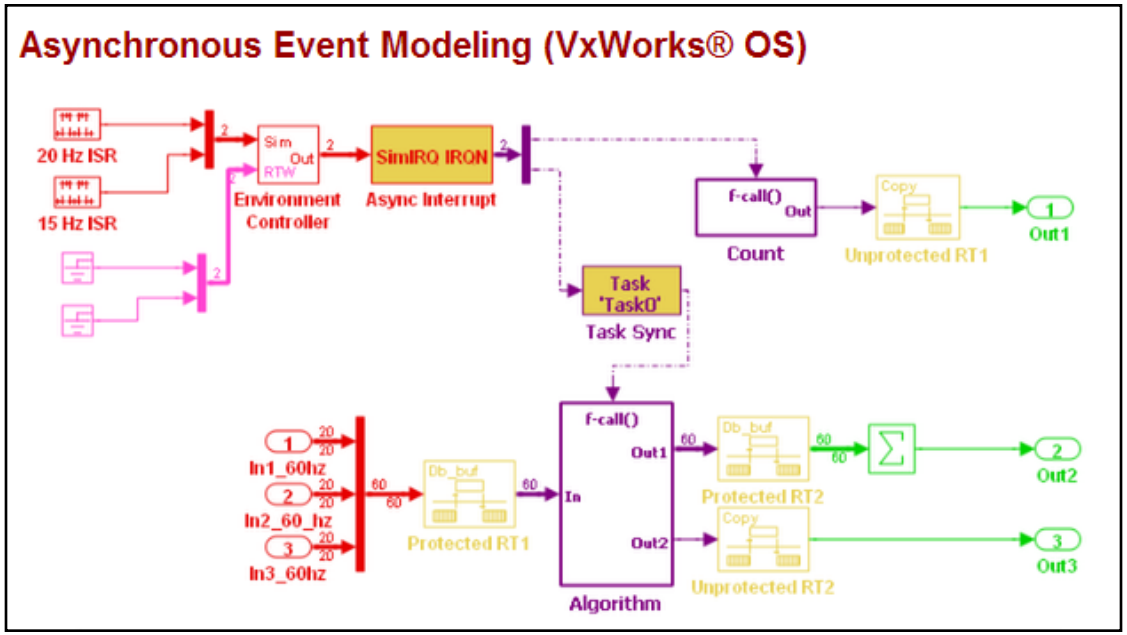
Real-Time Executable

Notes:
- **Complete executable generated with integration with RTOS**
- **Automated build via project creation or makefile-based**



VxWorks Example

- Example of Real-Time Operating System (RTOS) integration
  - Files are customizable to support integration with another RTOS
- Employs rate grouping
  - Each rate is grouped into a separate function
- Uses customizable main file feature
- Provides demos for
  - Single rate, single task
  - Multi rate, single task
  - Multi rate, multiple task

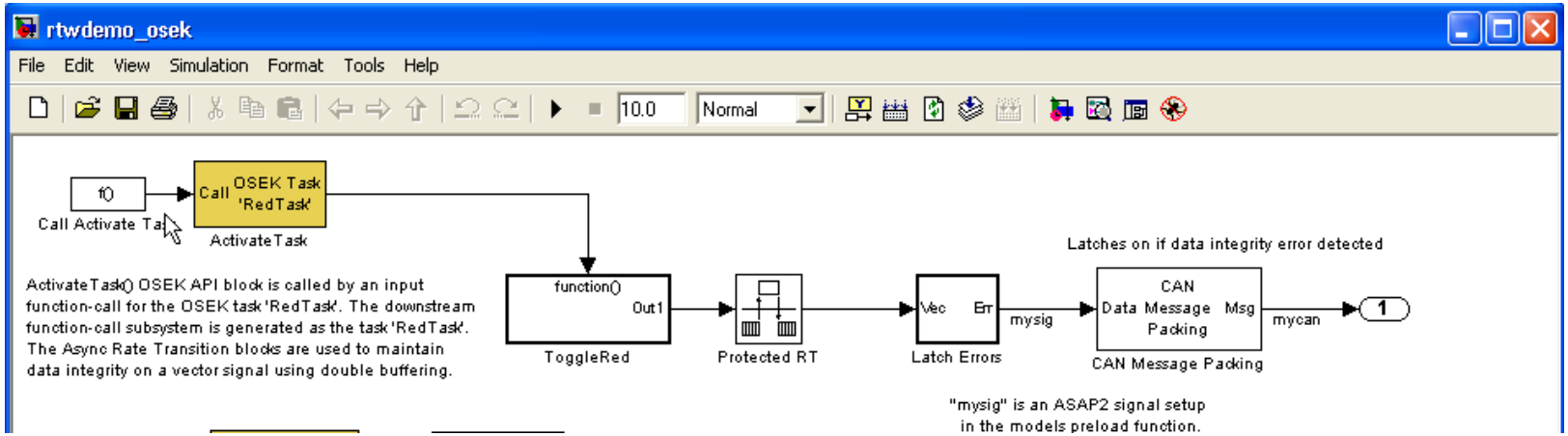


>> rtwdemo_async

>> rtwdemo_mrmtos

OSEK/VDX RTOS Demo Model

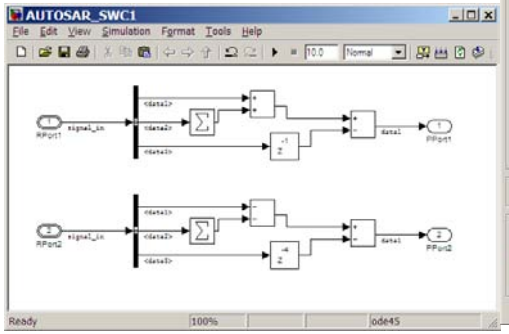
- OSEK/VDX is an RTOS used for some automotive systems
- OSEK/VDX demo includes:
  - Blocks and S-functions for `SetAlarm` and `ActivateTask` APIs
  - `main.c` and Implementation (OIL) file
  - CAN messaging
- Demo is starting point
  - End users can customize for their OSEK

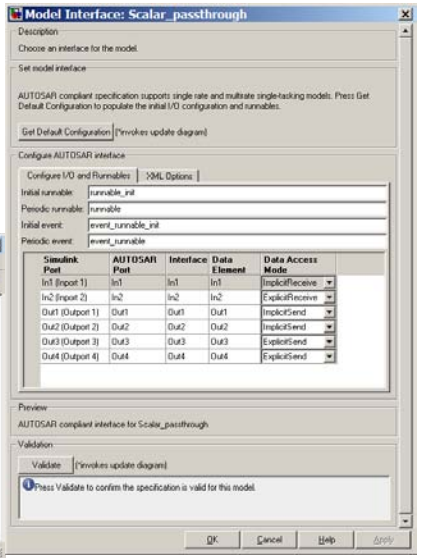



```
>> rtwdemo_osek
```

AUTOSAR Code Generation

- AUTOSAR is a run time environment for automotive applications
- Real-Time Workshop Embedded Coder provides an AUTOSAR target
- Generates AUTOSAR compliant functions (Runnables)
- Generates AUTOSAR XML descriptions for integration with AUTOSAR authoring and RTE generation tools (RTE) tools





Export/Code Gen

<xml>
</xml>

<xml>
</xml>

```
void runnable(void) {
Rte_read_p_d(&indata);
}
```

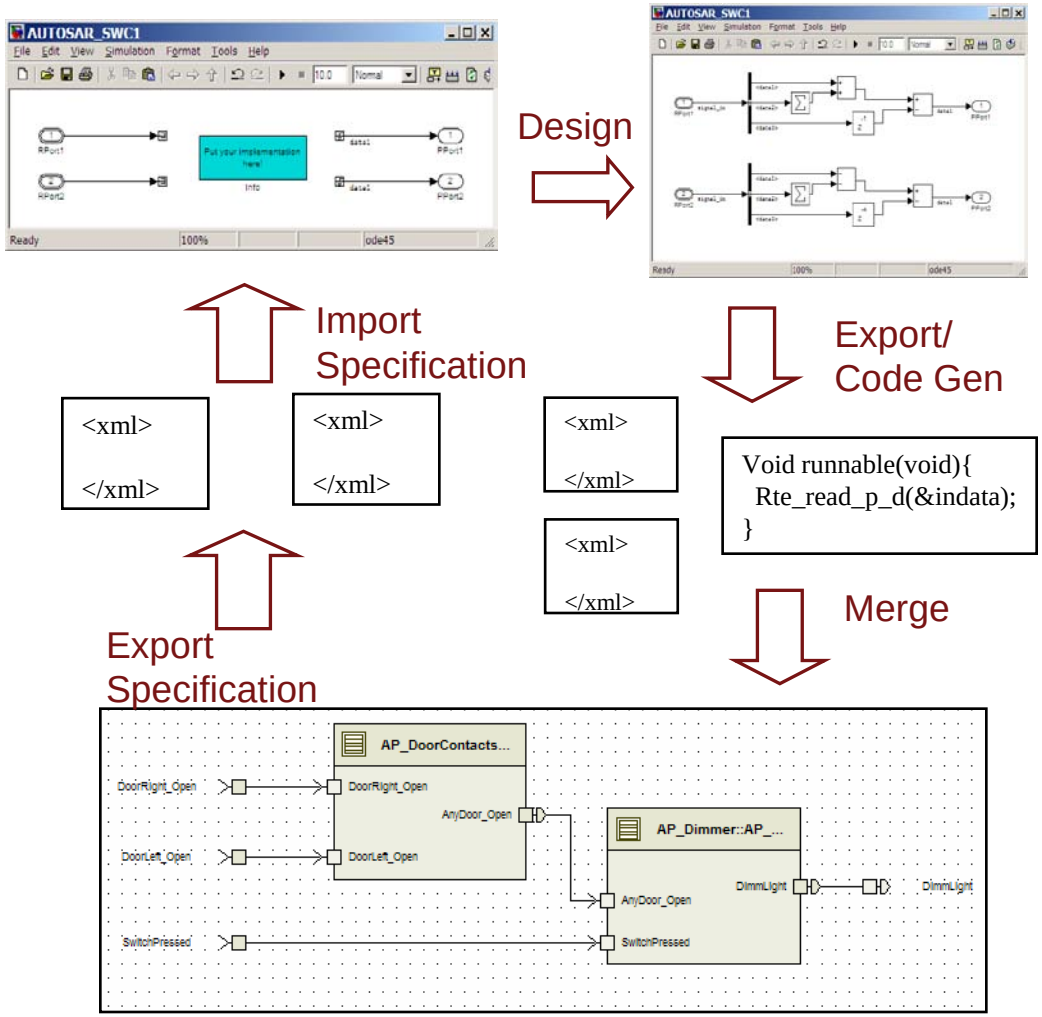
AUTOSAR Workflow

Problem

- Automotive engineers want to use Simulink as part of an AUTOSAR workflow and tool chain

Solution

- Enable import/export of AUTOSAR software component XML files
- Allow information to be merged back into AUTOSAR authoring tools such as Vector DaVinci products



AUTOSAR System Authoring Tool (e.g., DaVinci)

AUTOSAR Target – Multiple Runnables

Problem

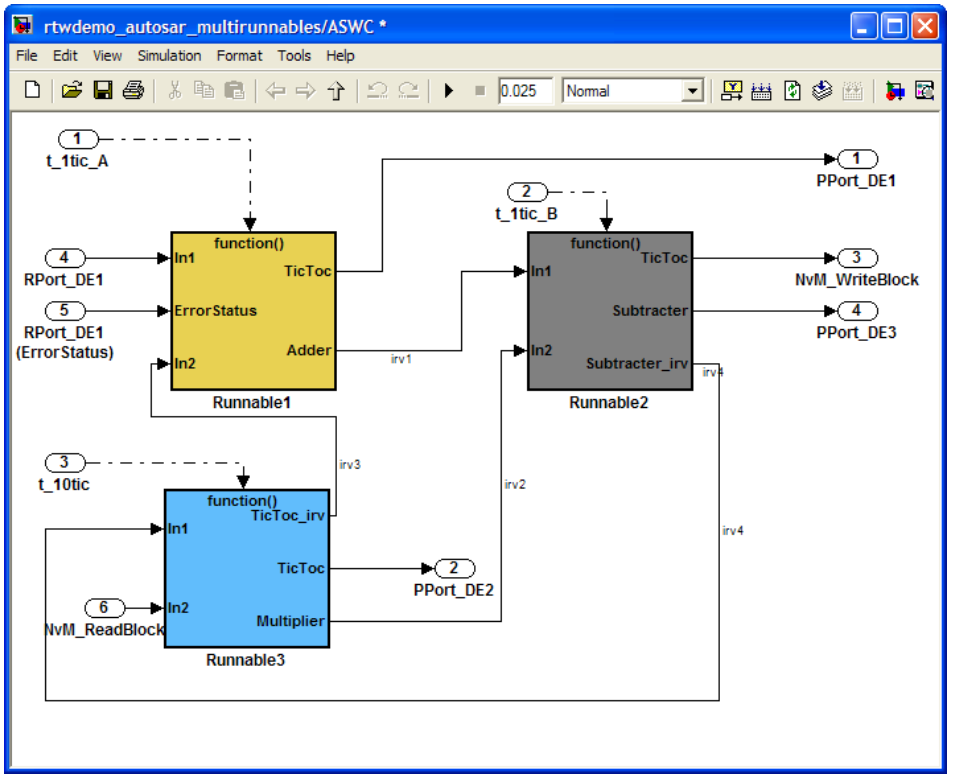
- Need to combine related functionality at multiple rates, as runnables in a single software component

Solution

- Virtual subsystem mapped to AUTOSAR Atomic Software Component
- AUTOSAR runnables modeled as function-call subsystems
- InterRunnableVariables are created to communicate data between runnables to ensure data integrity

Benefit

- Clean mapping of AUTOSAR runnable concept to Simulink
- Support automotive production code generation needs with AUTOSAR



>> rtwdemo_autosar_multirunnables

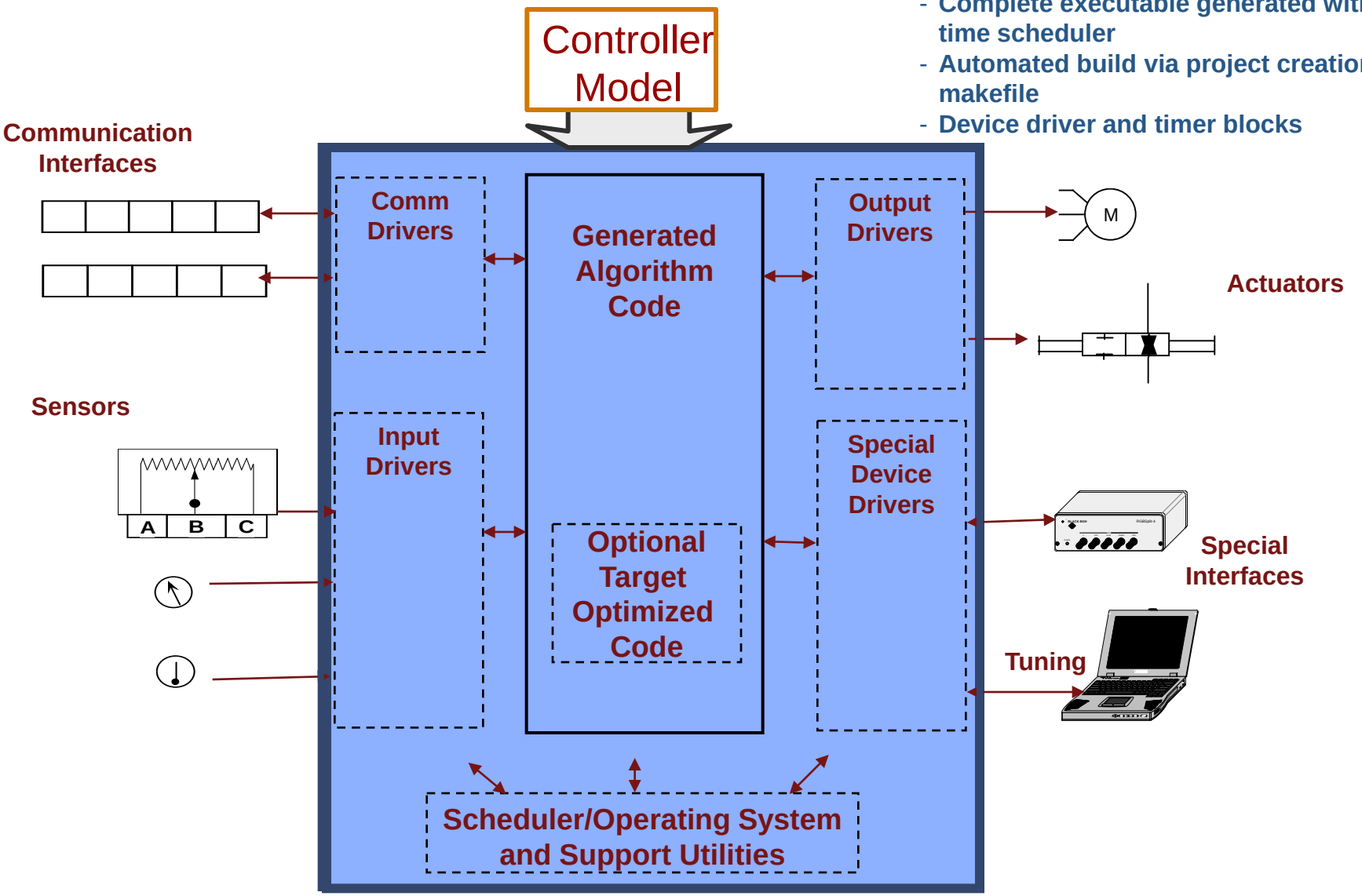
Target Integration Options

- Algorithm Export
  - Most popular option for production organizations
- Real-Time Target Support Examples
  - Provided with Real-Time Workshop Embedded Coder
- **Add-on Link and Target Products**
  - From MathWorks
  - From third parties
- Build your own link or target (Workshop Focus)
  - Using documented procedure and APIs

Production Code Generation and Integration

Full Embedded Target

Notes:
- Complete executable generated with real-time scheduler
- Automated build via project creation or makefile
- Device driver and timer blocks



Link and Target MathWorks Products

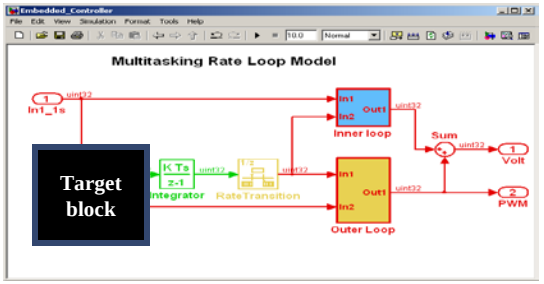
Embedded IDE Link (for IDEs)

- Makes it easy to verify generated and compiled object code using compiler toolchains (IDEs):
  - Altium TASKING
  - Eclipse
  - Green Hills MULTI
  - TI Code Composer Studio
  - Analog Devices VisualDSP++
  - HDL tool links

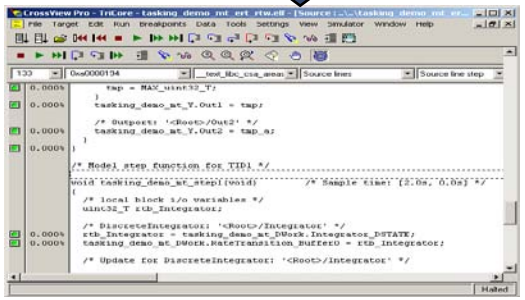
Target Support Package (for embedded processors)

- Makes it easy to deploy generated code on particular microprocessors including:
  - ARM
  - Freescale MPC5xx
  - Infineon C166 family
  - STMicroelectronics ST10 family
  - TI C2000 and C6000 families
- Targets are add-ons links

Simulink



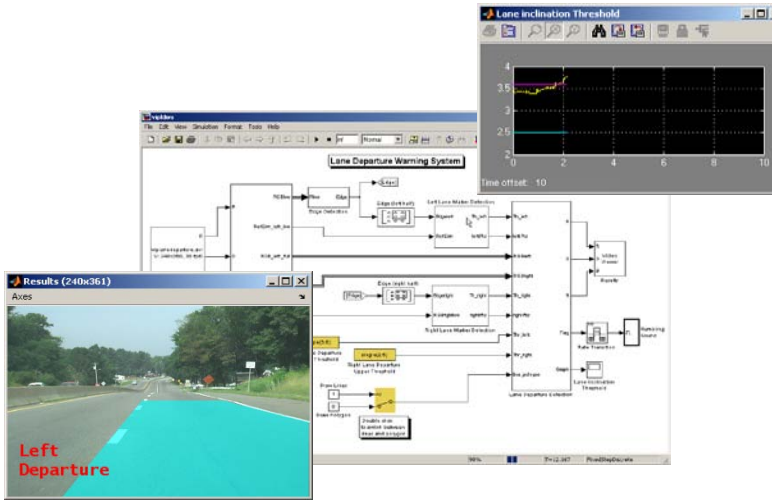
IDE (e.g., TASKING) **(Link)**



Target (e.g., ST10) **(processor)**

Embedded IDE Link Product Features

- MATLAB
  - APIs to analyze data and debug code generated automatically or by hand
- Simulink
  - Processor-in-the-loop (PIL) testing to verify object code using models
- Real-Time Workshop Embedded Coder
  - Automatic generation and download of code to instruction set simulator or MCU/DSP hardware



Target Support Package Product Features

- Automatic build and download
- Preconfigured to run with popular evaluation board(s)
- Execution on bare-board target using real-time scheduler
- Target optimizations
  - Basic math operator replacement (e.g., add, subtract)
  - Signal processing filters (e.g., FFT, IIR)
  - Bit-address memory
  - …
- Device driver blocks
  - A/D, D/A
  - DI, DO
  - Timer Interrupts
  - …

MathWorks Links and Targets Comparison

	Embedded IDE Link	Target Support Package
Processor Support	Support many processors supported by the IDE	Support select processors using one or more IDEs
Purpose	Primarily used for production software verification	Traditionally used for on-target rapid prototyping; Emerging use in production applications
Engineers	Typically used by software engineers in production	Typically used by systems engineers or advanced production
Programs	Work well for existing and new vehicle programs	Work well for new vehicle programs
Requirements	Require Real-Time Workshop Embedded Coder (for PIL)	Requires Embedded IDE Link product
Provider	Built by MathWorks and partners	Built by MathWorks and partners

Third Party Link and Target Products

The MathWorks™
Accelerating the pace of engineering and science

Home | Select Country | Contact Us |

Products & Services | Solutions | Academia | Support | User Communi

Product Overview
- Description
- Function List
- Block List
- Supported Hardware
- Demos and Webinars
- Related Products
- System Requirements
- Latest Features

Support & Training
- Product Support
- Documentation
- Downloads & Trials
- Consulting

Other Resources
- Technical Literature
- User Stories
- Related Books
- Free industry usage guides

Real-Time Workshop Embedded Coder 5.4

Supported Hardware

Real-Time Workshop Embedded Coder supports any hardware through portable ANSI/ISO C/C++ code. You specify data types for your processor using built-in support or by entering them manually. Additional support is provided by MathWorks link and target products or third-party partner products.

Complete the form below the table to request additional processor support.

Mouse over any of the column headings to see a description.

Processors	C/C++ Support	Built-In Data Type Support	Link Products	Target Products	Third-Party Products
All processors	✔				
AMD® K5®/K6®/Athlon®	✔	✔		xPC Target™	Various
Analog Devices® Blackfin®	✔	✔	Embedded IDE Link™		
Analog Devices® SHARC®	✔	✔	Embedded IDE Link™		
Analog Devices® tigerSHARC®	✔	✔	Embedded IDE Link™		
ARM® Compatible 7/8/9	✔	✔	Embedded IDE Link™	Target Support Package™	GAIO
Atmel® AVR	✔	✔			
Freescale™ 68332	✔	✔			
TI's C2000™	✔	✔	Embedded IDE Link™	Target Support Package™	
TI's C5000™	✔	✔	Embedded IDE Link™	Target Support Package™	
TI's C6000™	✔	✔	Embedded IDE Link™	Target Support Package™	
TI's MSP430	✔	✔			
TI's OMAP™	✔		Embedded IDE Link™		

Select the manufacturer to see a full list of supported hardware

(Partial List Shown)

Request Additional Hardware Interface Support

*Indicates Required Information

*Hardware Manufacturer

Comments

*E-mail

We will not sell or rent your personal contact information. See our privacy policy for details.

Submit

www.mathworks.com/products/rtwembedded/supportedio_rtwembedded.html

Target Support Options by Processor Family

Processors		C/C++ Support	Built-In Data Type Support	MathWorks Link Products	MathWorks Target Products	Third-Party Support
All Processors		√				
AMD®	K5®, K6®, Athlon®	√	√		√	√
Analog Devices®	Blackfin®, SHARC®, tigerSHARC®	√	√	√		
ARM®	Compatible 7/8/9	√	√	√	√	√
Atmel®	AVR	√	√			
Freescale™	Power Architecture generic 32-bit support	√	√			
	MPC7400	√	√	√		
	MPC5500	√	√	√		√
	MPC500	√	√		√	√
	HC(S)12	√	√			√
	68332, ColdFire®, 68HC11, 68HC08	√	√			
	DSP563xx (16-bit mode)	√	√	√		
Fujitsu®	16LX	√				√
Infineon®	TriCore®	√	√	√		
	C16x/ XC16x	√	√	√	√	
Intel®	8051 Compatible	√	√	√		
	x86/Pentium®	√	√		√	√
Microchip®	dsPIC, PIC18®	√	√			√
NEC®	V850	√	√	√		√
Renesas®	SH-2/3/4	√	√			√
	H8/H8S, M32R	√				√
	M16C	√	√	√		
	RC8/Tiny	√	√	√		
SGI®	UltraSPARC® IIi	√	√			
STMicroelectronics®	ST10, Super10	√	√	√	√	
Texas Instruments®	C6000™	√	√	√	√	
	C5000™, MSP430	√	√	√		
	C2000™	√	√	√	√	
	OMAP™	√		√		

See http://www.mathworks.com/products/rtwembedded/supportedio.html for the complete supported list

Target Integration Options for S12x

- Algorithm Export
  - Customers have successfully used this approach
    - See Motorola Automotive user story based on HC12
- Add-on Target Products From MathWorks
  - Not available
- Add-on Target Products From Third Parties
  - Mototron offers Motohawk for a full SW/HW system solution
  - SimuQuest offers QuantiPhi for a feature-rich target blockset
  - Other options not in MathWorks Connections program
    - Unis offers Processor Expert for a target blockset
- Build your own target options
  - Review Developing Embedded Targets guide in Real-Time Workshop Embedded Coder documentation
  - Attend this free, half day, Developing Embedded Targets Workshop
  - MathWorks consulting can extend the S12x Baseline example target

Introduction

- Simulink code generation
- Class Exercise #1
- Target integration options
- **S12x Example Target**

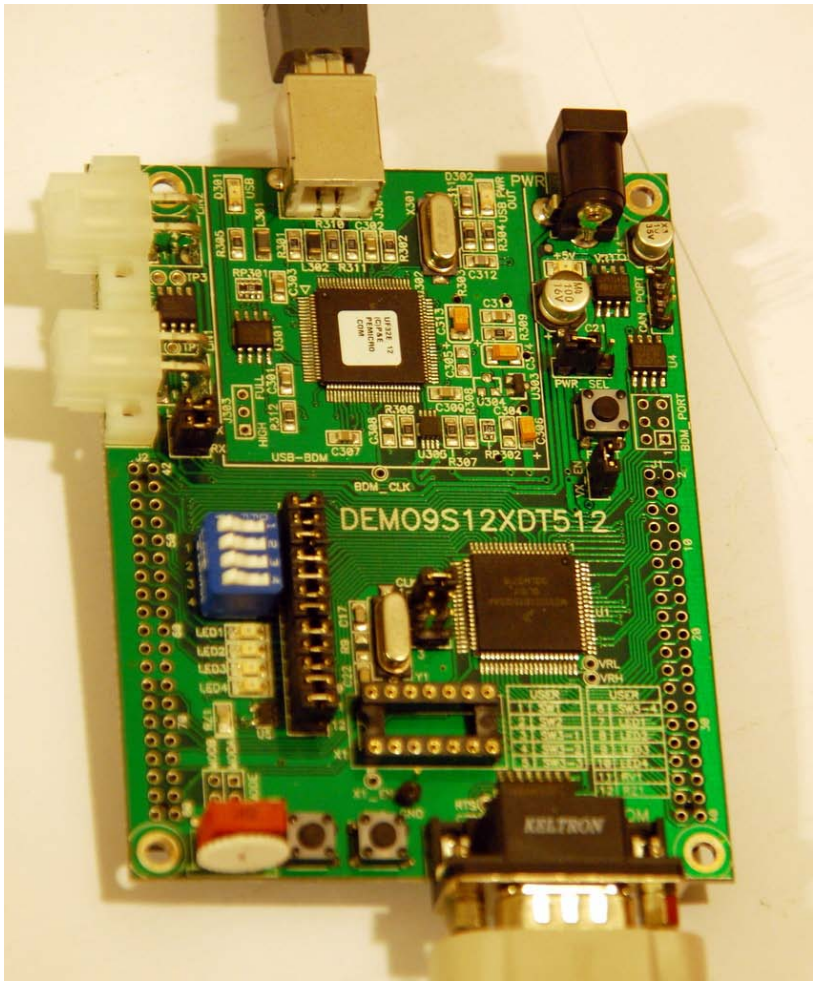
S12x Example Target

- Goal
  - Provide an baseline target with example I/O blocks that can be used as a basis for end user to develop custom targets
- Third Party products
  - Cosmic compiler and ZAP debugger (free download for programs less than 4KB)
  - Evaluation board from Freescale
    - DEMO9S12XDT512
- Block Support
  - Digital In/Out
  - Analog In
  - Button In
  - LED
  - Serial Communication TX/RX
  - Interrupt Timer
- Supports PIL
- CodeWarrior support is available
- Contact MathWorks for updates
  - mytarget@mathworks.com

DEMO9S12XDT512E-ND Board

Features

- 80-Pin QFP MC9S12XDT512CAA Microcontroller
- Selectable 4MHz Crystal and a provision for an Oscillator Module (Socketed)
- Built-in USB to BDM interface for in-circuit Debugging
- Provision for a BDM Connector for External In-Circuit Debugging
- Two LIN Connector with one transceiver
- One RS-232 Connector with transceiver
- Four User LEDs
- Four DIP-Switches
- Potentiometer for Analog Input
- 2 Push-buttons
- Reset Push-button



Agenda

I. Introduction (60 minutes)
II. Creating a Host Target (60 minutes)
III. Lunch
IV. Creating an Embedded Target (60 minutes)
V. Verifying an Embedded Target using PIL test (30 minutes)

Creating a Host Target

- **Create baseline target**
- Create blockset (if needed)

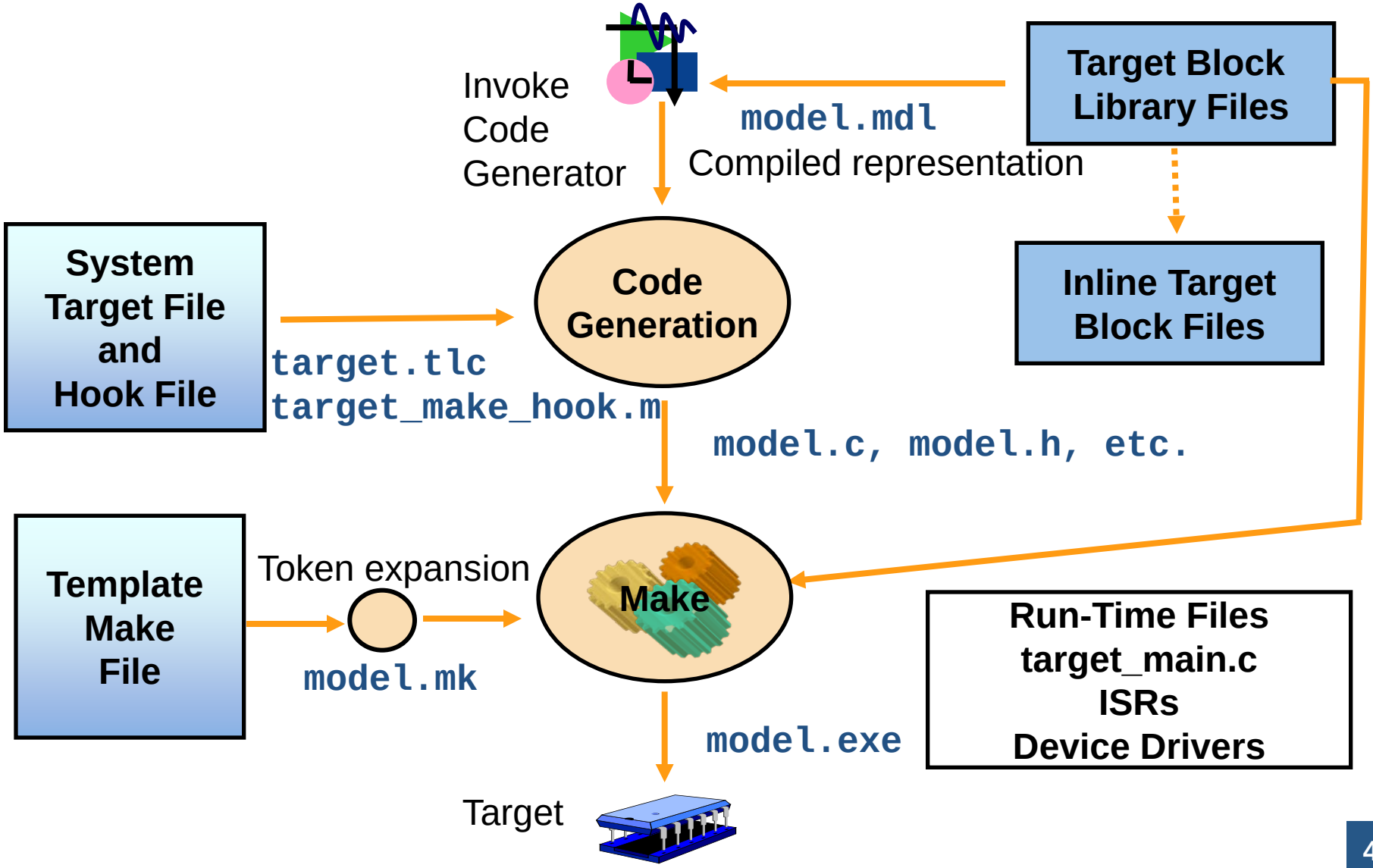
Note: This section introduces several file templates programmed in a script language, Target Language Compiler (TLC). Given the simple nature of the host target examples, a discussion of TLC at this point should not be necessary. However TLC will be introduced later during the Embedded Target section which uses TLC for some advanced file processing.

Students can look ahead if they wish to gain some background on TLC.

Creating a Baseline Target

- **Create baseline target**
  - **Develop target directory structure**
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Customize Main file
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- A class exercise to create a host target named *htgt* will be incorporated during each baseline target step.

Baseline Target Files



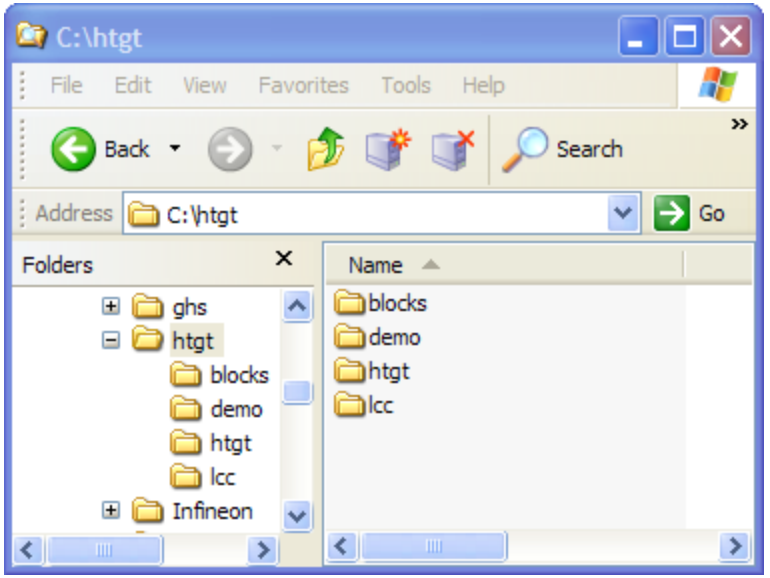
Example Target Directory Structure

- *targetname* – Target root directory
  - blocks – Contains files associated with the I/O devices
    - Source and executable files of I/O driver C model interface functions
    - ASCII files that inline the I/O driver S-functions
    - A library of I/O driver blocks
  - *compilername* – Contains files associated with the specific compiler
  - demo – Contains demos
  - *targetname* – Contains control files central to the target
    - System target file
    - Template makefile
    - File customization template that produces a main *targetname*_main.c
    - Hook file *targetname*_make_hook.m that intervene the build process to dispatch target specific actions
    - More files

Class Exercise 2a

Create your host target directory structure

- Create a base name for your target, *htgt*
- Create a directory structure for your target, *C:\htgt*
  - Place outside of MATLAB installation path
  - Include subdirectories: *blocks, lcc, demo, htgt*
  - Keep subdirectories empty of files
- Add *htgt* root directory with subfolders to MATLAB path



Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - **Create System Target file**
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Customize Main file
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

System Target File

- The system target file exerts overall control of the code generation stage of the target build process.
- The system target file serves the following purposes:
  - Supplies target information to the System Target File browser and entries to the target make command
  - Registers template variables that setup the code generator environment to control code format
  - Provides entry point to the code generation process
  - Generates user interface elements of target-specific options and inherits code generation options from another target
  - Provides a mechanism to automate setting of model configuration constraints
- The system target file is conventionally named after the target

System Target File Structure

Browser comments

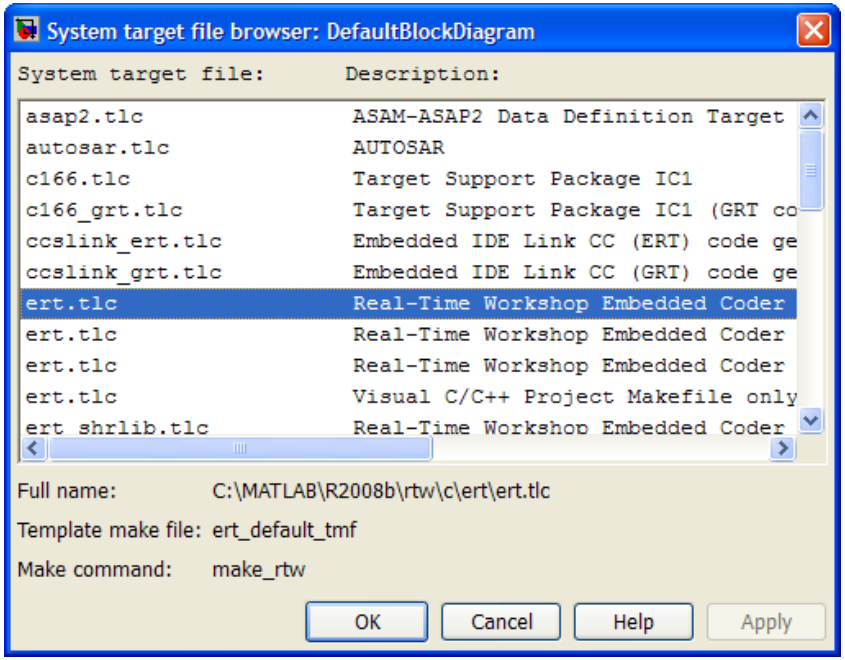
- **SYSTLC**: String that describes target
- **TMF**: Name of template makefile
- **MAKE**: Make command to use
- **EXTMODE**: Name of external mode interface file

TLC configuration variables

- **CodeFormat**: Code format
- **Language**: **C**, or **C++**,
- **TargetType**: **RT** or **NRT**

TLC program entry point

`RTW_OPTIONS` **section**



Note that ERT has 4 different Browser Descriptions

Example ERT System Target File (Top)

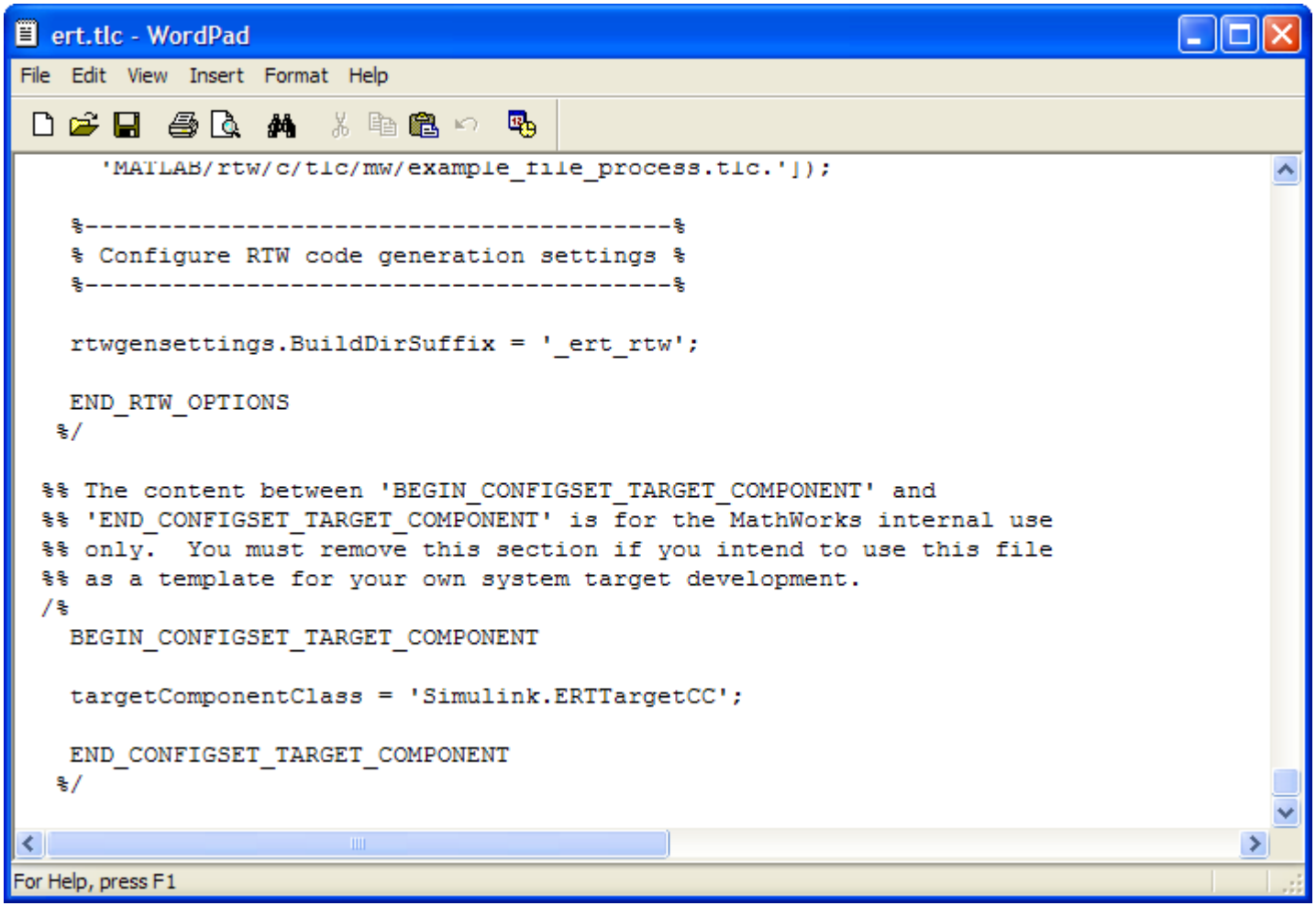
ert.tlc - WordPad

File Edit View Insert Format Help

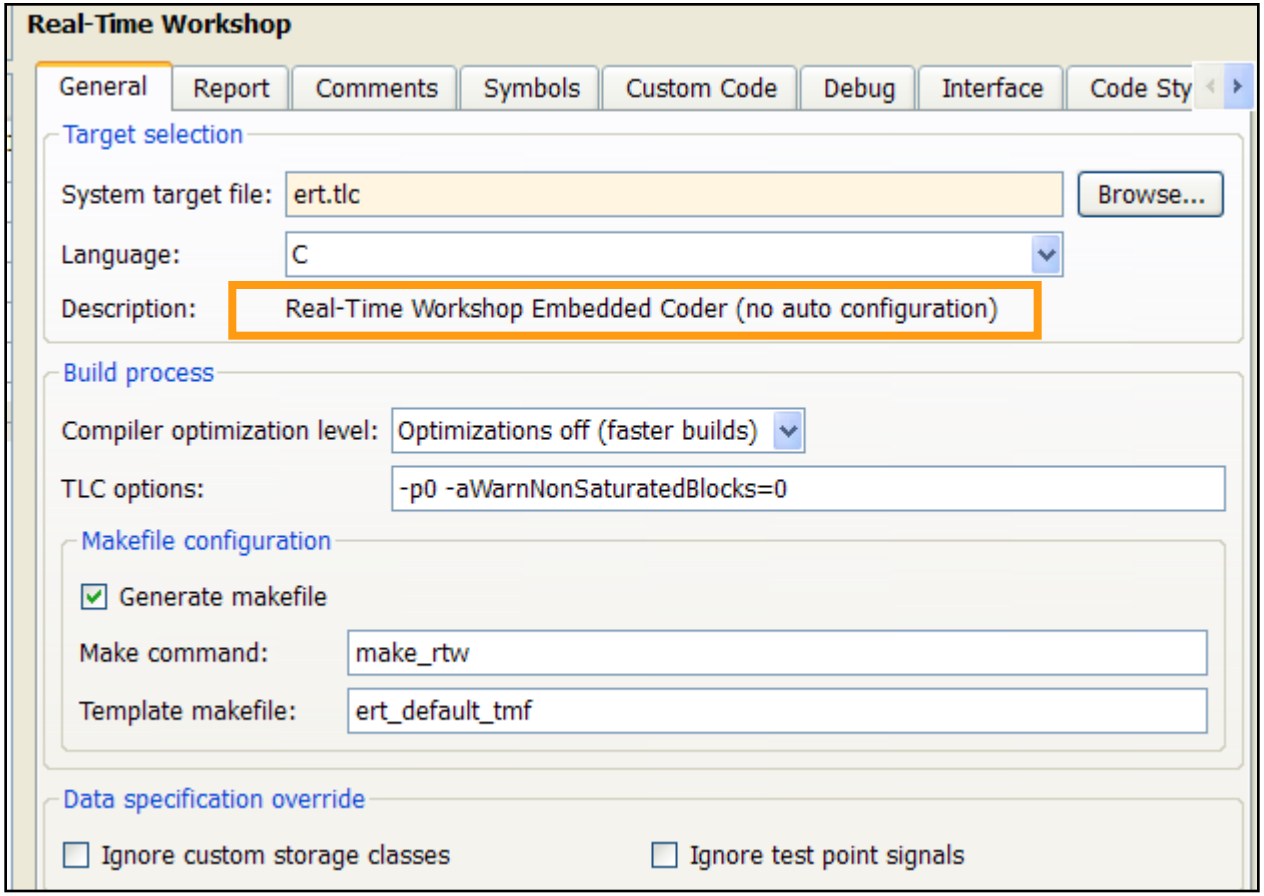
```
%% SYSTLC: Real-Time Workshop Embedded Coder (no auto configuration) TMF: ert_default_tmf MAKE: make_rtw \
%%    EXTMODE: ext_comm
%% SYSTLC: Real-Time Workshop Embedded Coder (auto configures for optimized fixed-point code) TMF: ert_defau
%%    EXTMODE: ext_comm
%% SYSTLC: Real-Time Workshop Embedded Coder (auto configures for optimized floating-point code) TMF: ert_de
%%    EXTMODE: ext_comm
%% SYSTLC: Visual C/C++ Project Makefile only for the Real-Time Workshop Embedded Coder\
%%    TMF: ert_msvc.tmf MAKE: make_rtw EXTMODE: ext_comm
%%
%% $RCSfile: ert.tlc,v $
%% $Revision: 1.57.4.14 $
%%
%% Copyright 1994-2008 The MathWorks, Inc.
%% Abstract: Embedded real-time system target file.
%%
%selectfile NULL_FILE

%assign CodeFormat = "Embedded-C"

%assign TargetType = "RT"
%assign Language   = "C"
%assign AutoBuildProcedure = !GenerateSampleERTMain
```

For Help, press F1

Example ERT System Target File (Bottom)



Example System Target File Selection



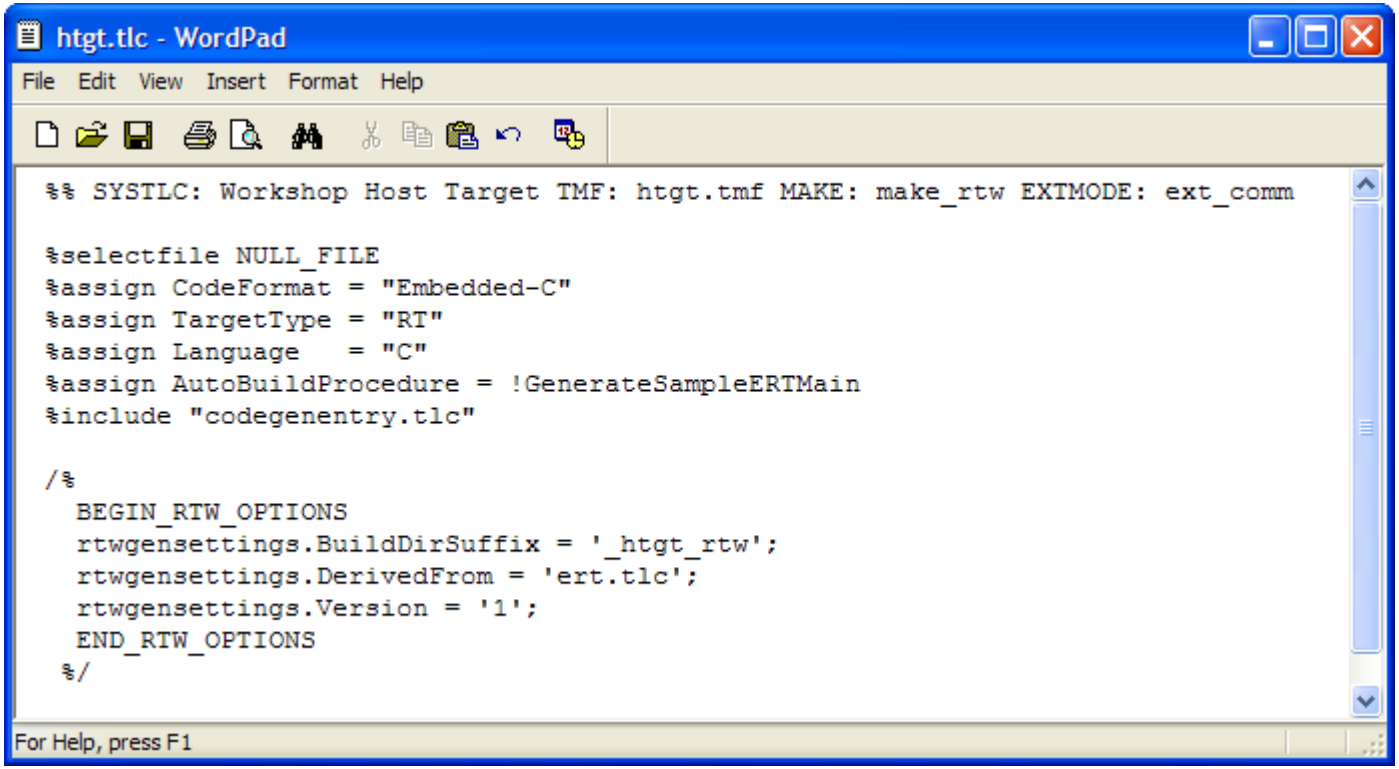
Creating System Target files

- Examine ert.tlc in matlabroot/rtw/c/ert folder
- Copy ert.tlc to your *targetname/targetname* subdirectory
- Create a simplified version tailored to your target needs

Student Exercise 2b

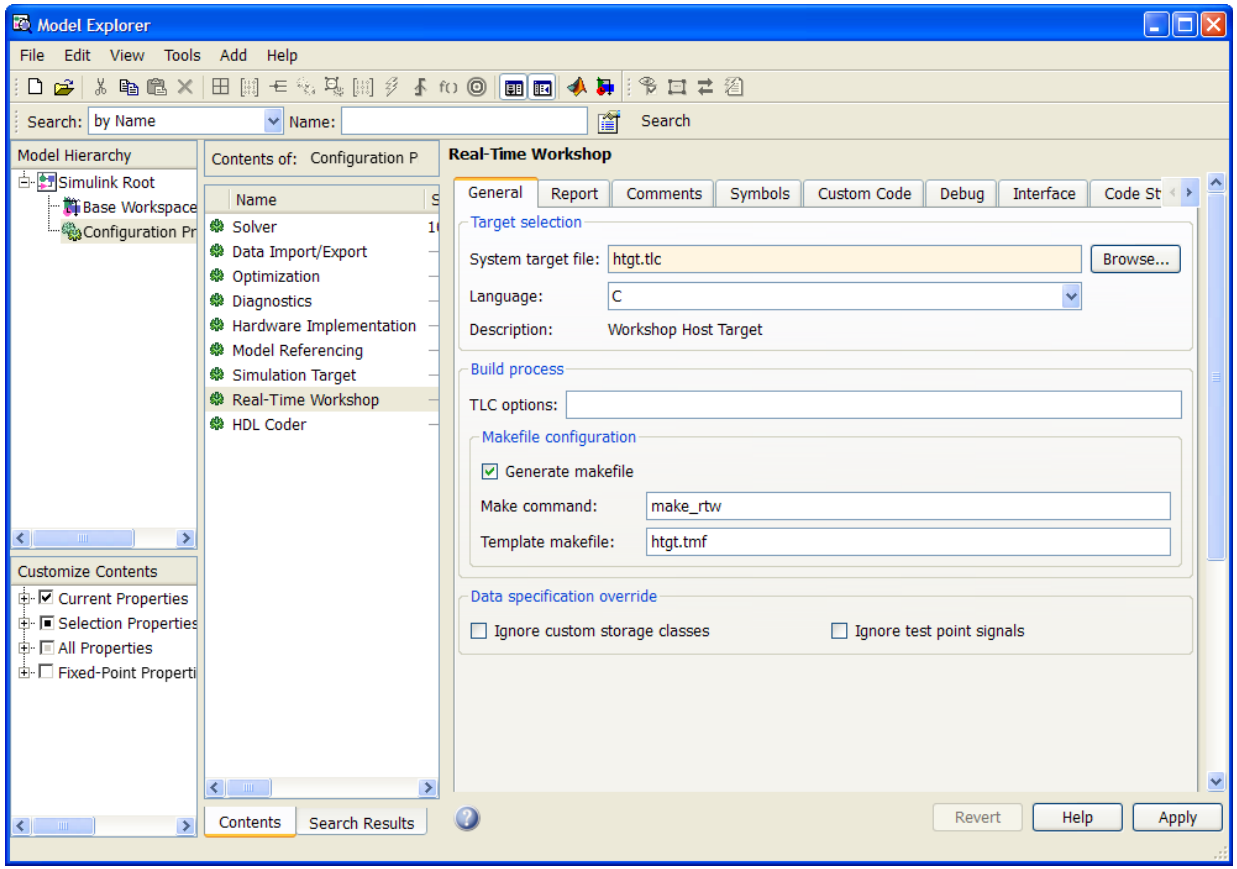
Create a new System Target File based on ERT

- Copy ert.tlc to htgt.tlc
  - `cp matlabroot/rtw/c/ert/ert.tlc c:/htgt/htgt/htgt.tlc`
- Edit htgt.tlc as shown below



Student Exercise 2b (cont.)

- Select *htgt* as the STF and confirm it matches the figure below
- Do not generate code because the *htgt* makefile has not been created



Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - **Customize Makefile**
  - Customize Hook Files for post processing, token additions
  - Customize Main file
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

The Make Process

- Make is the final phase of the code generation and build process where
  - Sources are compiled
  - Libraries are generated if necessary
  - Object code and libraries are linked to generate an executable
  - Executable is optionally loaded onto target hardware
- Two approaches to handle the make process
  - Generate a makefile from a template makefile via a mechanism called token expansion to work with a make utility such as GNU Make
  - Automate the creation of a project file via a hook file to work with an integrated development environment (IDE)
- This workshop uses makefiles

Template Makefiles

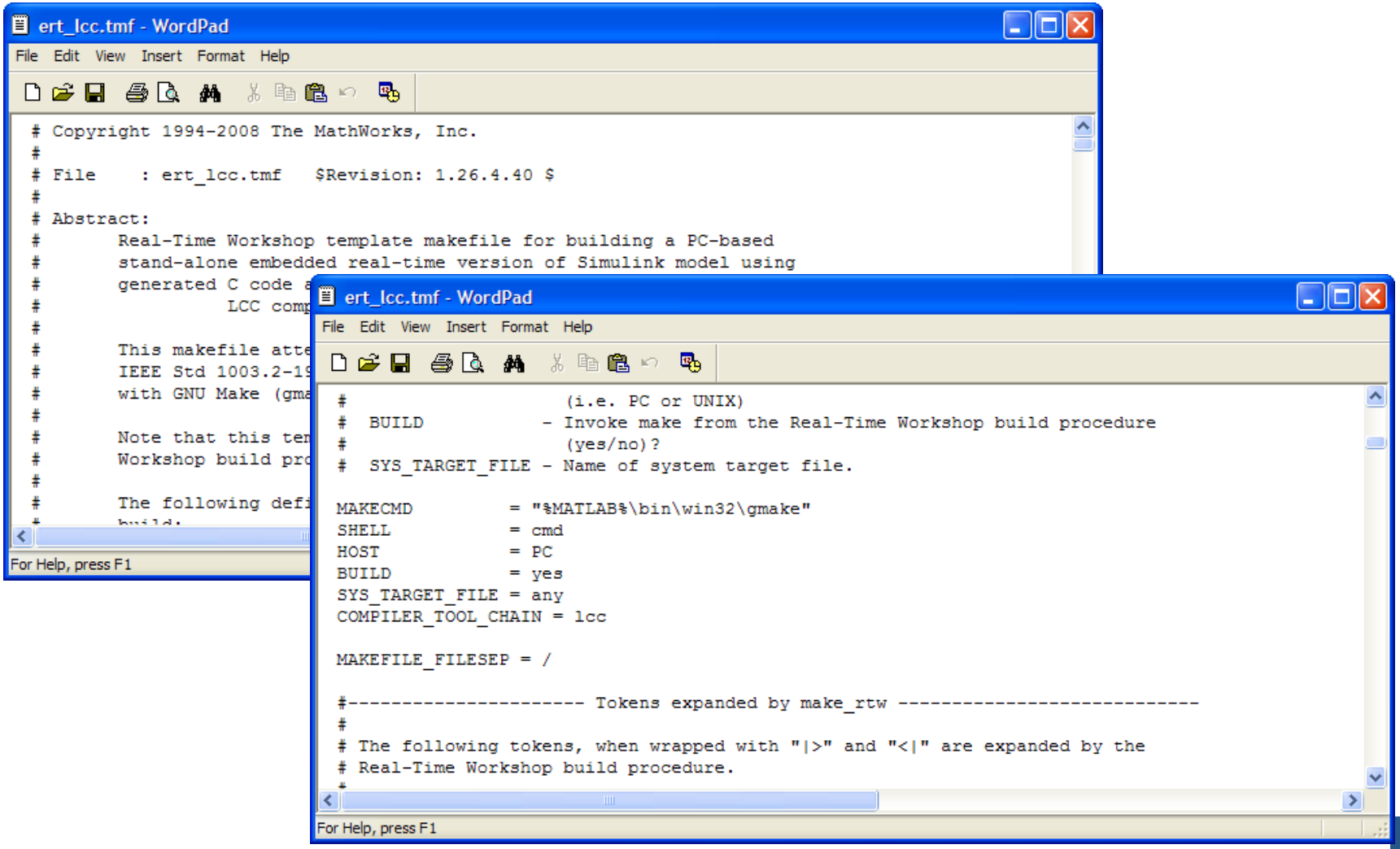
Customizable to generate makefiles specific to a target environment

Generates `model.mk`. The makefile:

- Is specifically tailored to compile and link generated code from your model
- Uses commands specific to your development system

Made up of statements containing *tokens*, to be expanded during the build process by `make_rtw`. For example `|>EXT_MODE<|` gets expanded to `1` or `0` depending on external mode support specification in the Real-Time Workshop software options.

Example ERT LCC Template Makefile



Creating Template Makefiles

- Examine a makefile in matlabroot/rtw/c/ert folder
- Copy the makefile to your *targetname/targetname* subdirectory
- Set the SYS_TGT_FILE to your System Target File
- Perform additional changes as appropriate for your target

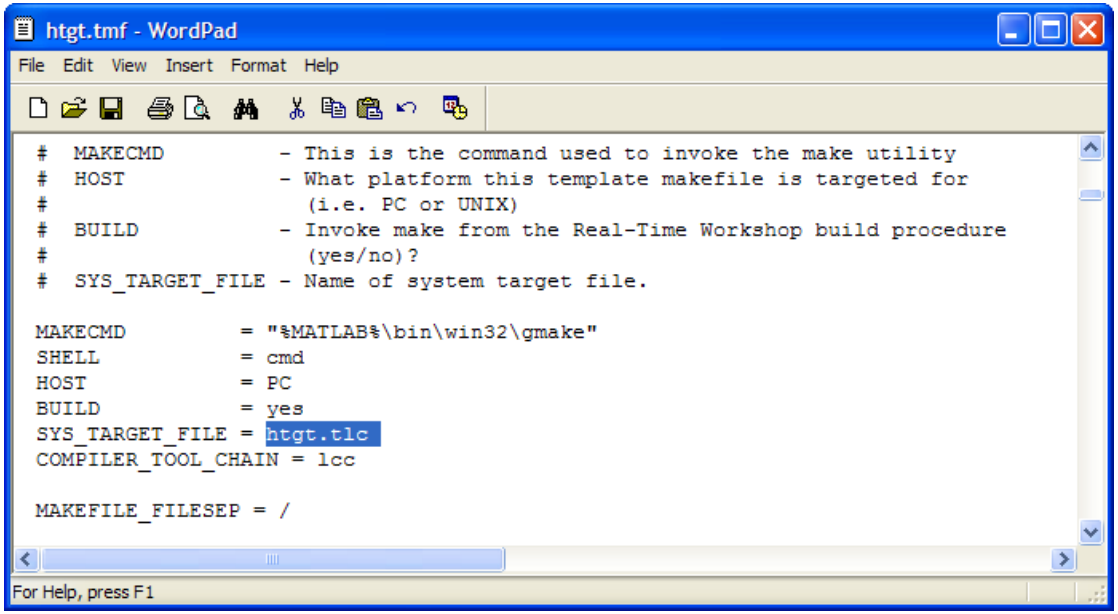
Student Exercise 2c

Create a new makefile based on ERT LCC

- Copy ert_lcc.tmf to htgt.tmf
  - `cp matlabroot/rtw/c/ert/ert_lcc.tmf c:/htgt/htgt/htgt.tmf`
- Create a minimal TMF

as shown in the figure

- Generate code for your simple model using your new target



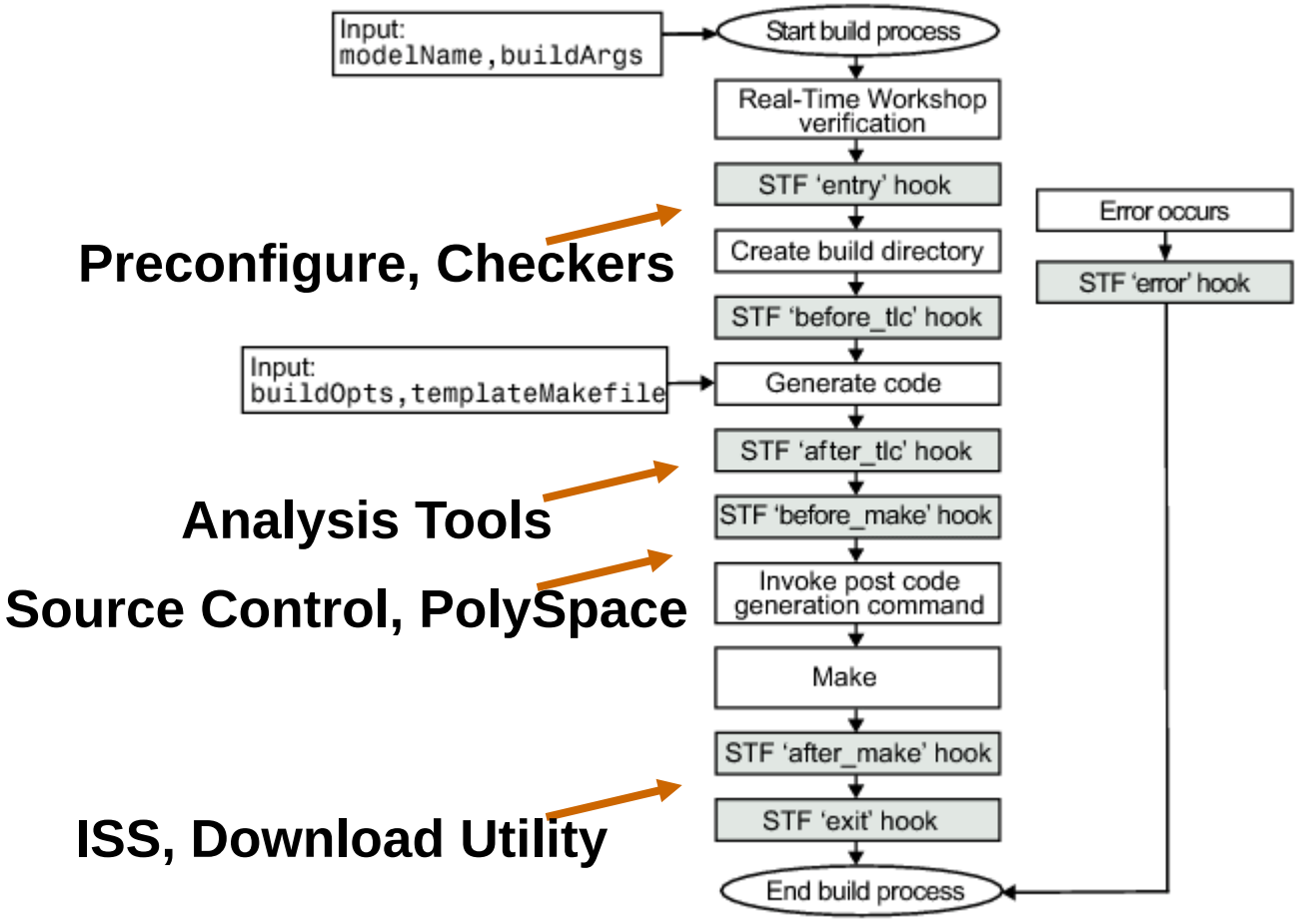
Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - **Customize Hook Files for post processing, token additions**
  - Customize Main file
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

Hook Files for post processing, token addition

- The build process lets you supply optional hook files that are executed at specified points in the build process.
- You can use hook files to add target-specific actions to the build process.
- An important hook file invoked during build is
  - *STF*_make_rtw_hook

Hook Files Entry Points During Build



Example ERT Make Hook File

ert_make_rtw_hook.m - WordPad

File Edit View Insert Format Help

```
function ert_make_rtw_hook(hookMethod,modelName,rtwroot,templateMakefile,buildOpts,buildArgs)
% ERT_MAKE_RTW_HOOK - This is the standard ERT hook file for the RTW build
% process (make_rtw), and implements automatic configuration of the
% models configuration parameters.  When the buildArgs option is specified
% as 'optimized_fixed_point=1' or 'optimized_floating_point=1', the model
% is configured automatically for optimized code generation.
%
% This hook file (i.e., file that implements various RTW callbacks) is
% called by RTW for system target file ert.tlc.  The file leverages
% strategic points of the RTW process.  A brief synopsis of the callback
% API is as follows:
%
% ert_make_rtw_hook(hookMethod, modelName, rtwroot, templateMakefile,
%                   buildOpts, buildArgs)
%
% hookMethod:
%   Specifies the stage of the RTW build process.  Possible values are
%   entry, before_tlc, after_tlc, before_make, after_make and exit, etc.
%
% modelName:
%   Name of model.  Valid for all stages.
%
% rtwroot:
%   Reserved.
%
% templateMakefile:
%   Name of template makefile.  Valid for stages 'before_make' and 'exit'.
%
% buildOpts:
%   Valid for stages 'before_make' and 'exit', a MATLAB structure
%   containing fields
%
%   modules:
```

For Help, press F1

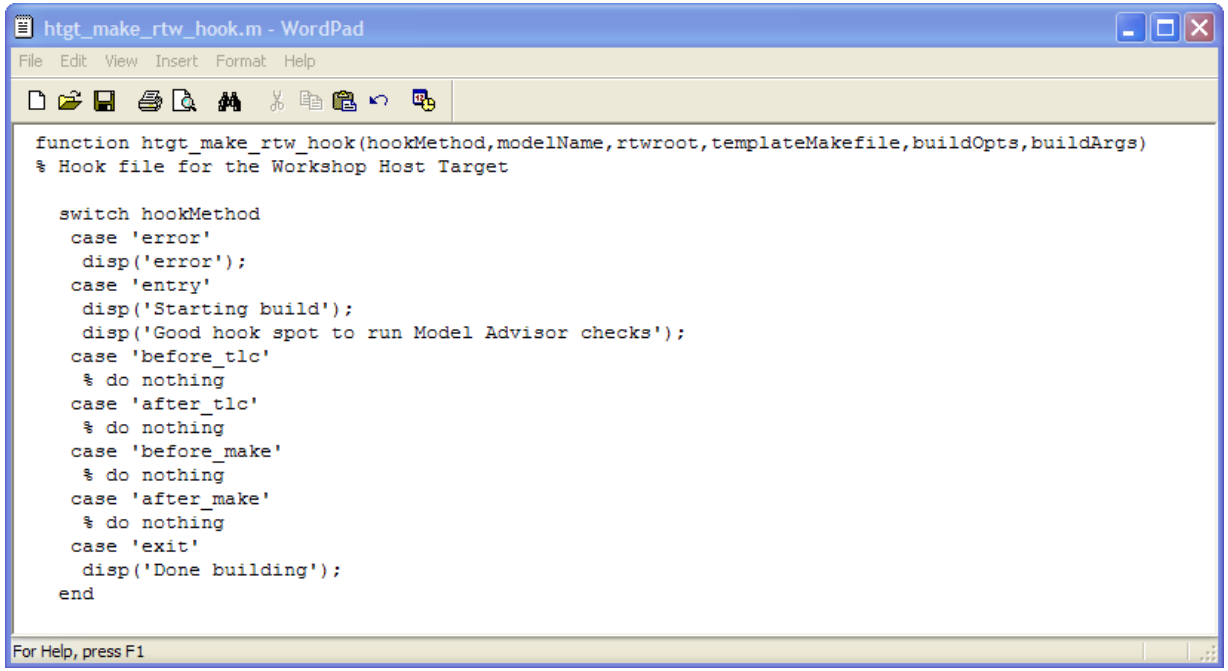
Creating Hook Files

- Examine a hook file in matlabroot/toolbox/rtw/targets/ecoder directory
- Copy the ert_make_rtw_hook.m to your *targetname/targetname* subdirectory
- Change the file prefix to use your System Target File
- Rename the ert_make_rtw_hook function within the file to match the filename.
- Implement the hooks that you require by adding code to the appropriate case statements within the switch hookMethod statement.

Student Exercise 2d

Create a Custom Hook File

- Copy ert_make_rtw_hook.m to htgt_make_rtw_hook.m
  - `cp matlabroot/toolbox/rtw/targets/ecoder/ert_make_rtw_hook.m c:/htgt/htgt/htgt_make_rtw_hook.m`
- Change hook file as shown below
- Generate code and observe MATLAB window message



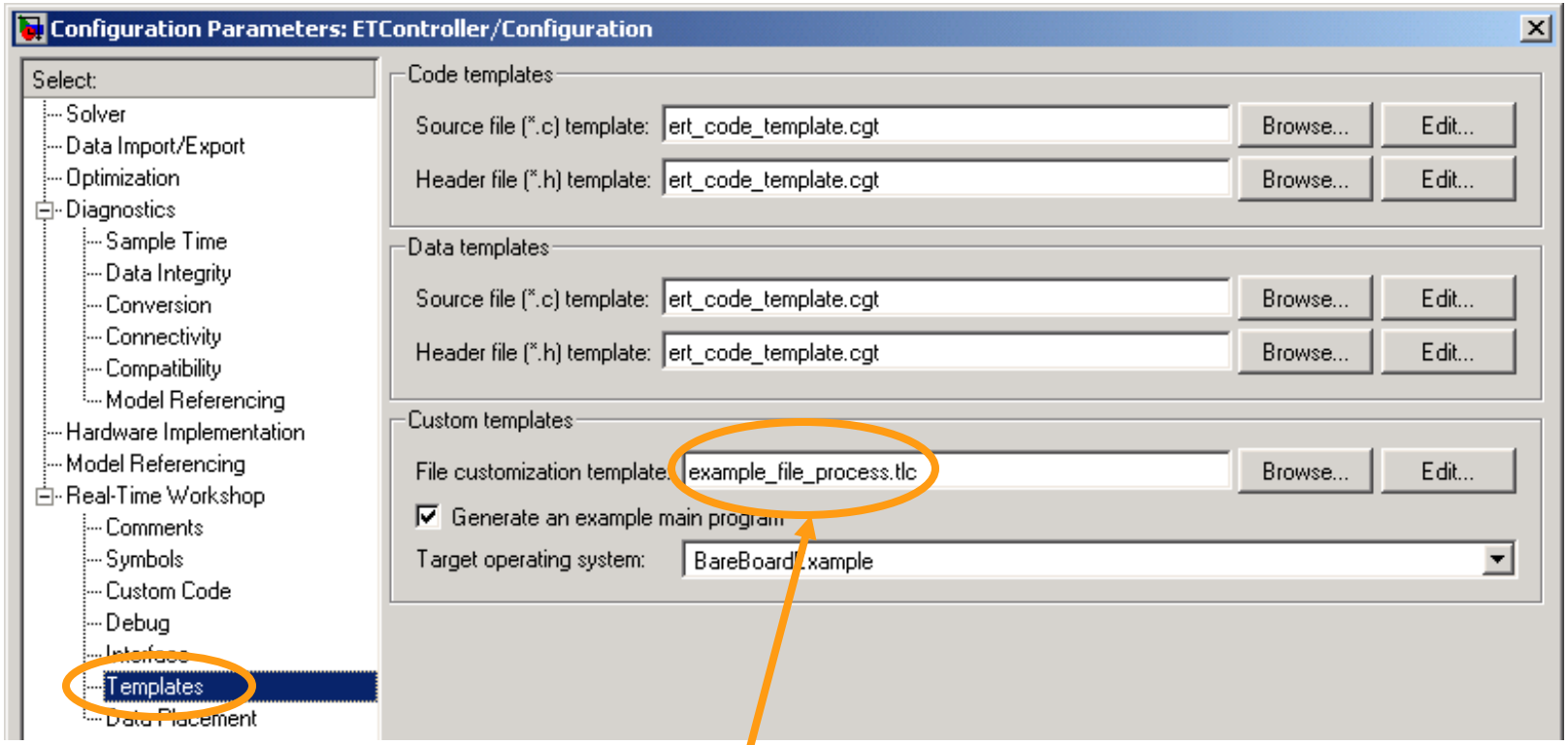
Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - **Customize Main file**
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

Hook Files for Generating Main

- Another hook file is used to create new or modified files including ert_main.c
- The hook file to do this is:
  - matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc
- It is invoked during code generation as shown on the following slide
  - However, it's contents are ignored due to a commented out line at the start of the file

Generate Example Main



File template can generate a custom main program

Example File Process Hook

example_file_process.tlc - WordPad

File Edit View Insert Format Help

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% $RCSfile: example_file_process.tlc,v $
%% $Revision: 1.1.6.3 $
%% $Date: 2006/10/10 02:35:49 $
%%
%% Abstract:
%%   Example Real-Time Workshop Embedded Coder custom file processing template.
%%
%%   Note: This file can contain any valid TLC code, which Real-Time Workshop
%%   executes just prior to writing the generated source files to disk.
%%   Using this template "hook" file, you are able to augment the generated
%%   source code and create additional files.
%%
%% Copyright 1994-2006 The MathWorks, Inc.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%selectfile NULL_FILE

%%  Uncomment this TLC line to execute the example
%%   ||    ||
%%   ||    ||
%%   \/    \/
%%  %assign ERTCustomFileTest = TLC_TRUE

%if EXISTS("ERTCustomFileTest") && ERTCustomFileTest == TLC_TRUE

  %% Need to set the template compliance flag before you can use the API
  %<LibSetCodeTemplateComplianceLevel(1)>
```

For Help, press F1

Uncomment to execute

Example File Process Hook (cont.)

Modify here to call your own single rate, single tasking template main file

example_file_process.tlc - WordPad

File Edit View Insert Format Help

```
/* #include "mytables.h" */
%closefile tmpBuf

%<LibSetSourceFileSection(modelC,"Includes",tmpBuf)>

%% Create a simple main.  Files are located in MATLAB/rtw/c/tlc/mw.

%if LibIsSingleRateModel() || LibIsSingleTasking()
  %include "bareboard_srmain.tlc"
  %<FcnSingleTaskingMain()>
%else
  %include "bareboard_mrmain.tlc"
  %<FcnMultiTaskingMain()>
%endif
```

For Help, press F1

The Example File Process Hook is well document and illustrates a number of file generation features in additional to main files.

Template Main File

- ert_main.c is generated by a template file that is intended to modified for the target environment
- Several template main files are available including support for single and multi rate execution at matlabroot/rtw/c/tlc/mw
  - bareboard_srmain.tlc
  - bareboard_mrmain.tlc
- End users should modify the templates such that rt_OneStep can handle interrupts and invoke model_initialize, model_step, model_terminate appropriately
- The default generated code for rt_OneStep using a multirate scheduler is shown on next two slides
- The simpler single rate rt_OneStep is used for our host target example in the slides that follow

Example Scheduling (Multirate, Multitask)

C:\class\work\filter_multirate_ert_rtw\ert_main.c

File Edit Text Cell Tools Debug Desktop Window Help

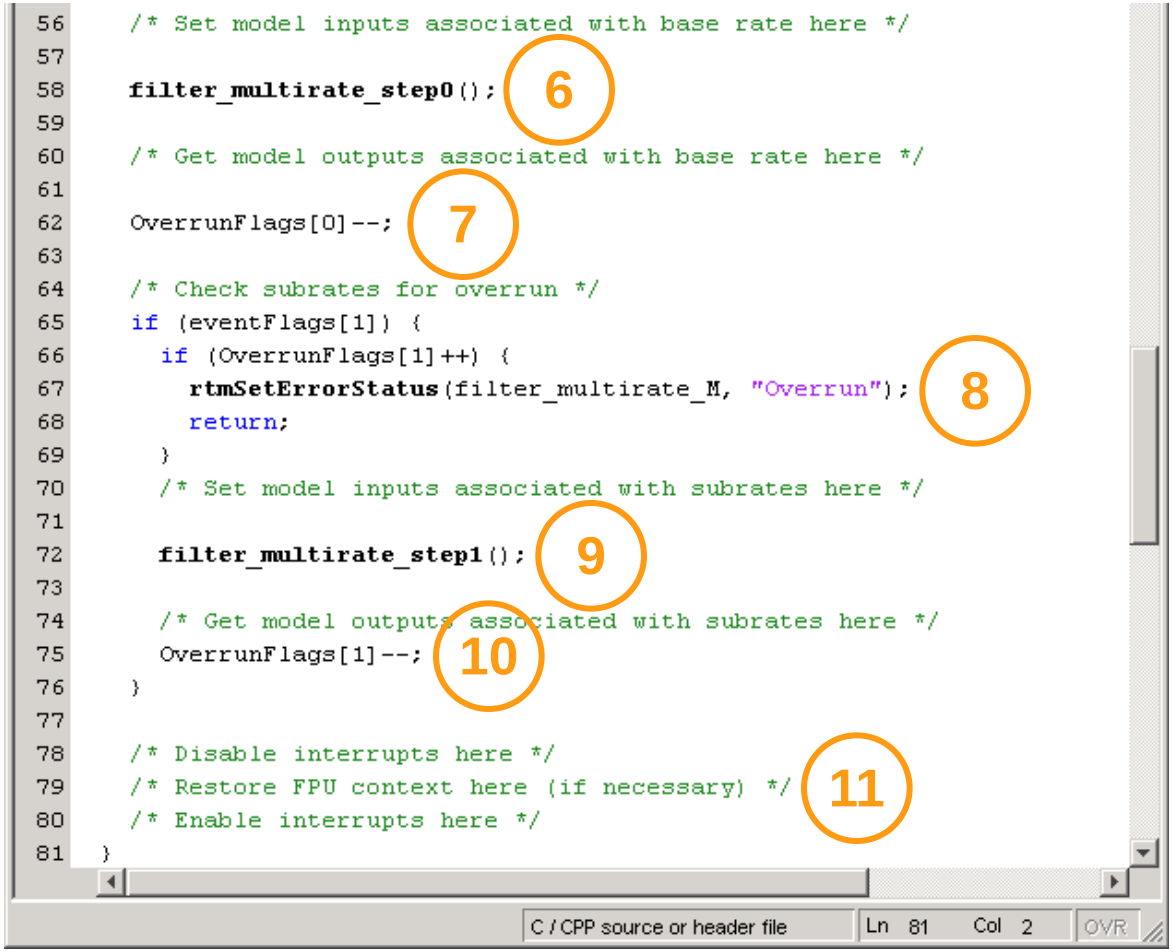
Stack: Base

```
18
19  static boolean_T OverrunFlags[2] = {0,0};

31  void rt_OneStep(void)
32  {
33    boolean_T eventFlags[2];       /* Model has 2 rates */
34
35    /* Disable interrupts here */
36
37    /* Check base rate for overrun */
38    if (OverrunFlags[0]++) {
39      rtmSetErrorStatus(filter_multirate_M, "Overrun");
40      return;
41    }
42
43    /* Save FPU context here (if necessary) */
44    /* Re-enable timer or interrupt here */
45
46    /*
47     * For a bare-board target (i.e., no operating system), the rates
48     * that execute this base step are buffered locally to allow for
49     * overlapping preemption.  The generated code includes function
50     * filter_multirate_SetEventsForThisBaseStep() which sets the rates
51     * that need to run this time step.  The return values are 1 and 0
52     * for true and false, respectively.
53     */
54    filter_multirate_SetEventsForThisBaseStep(eventFlags);
```

1
2
3
4
5

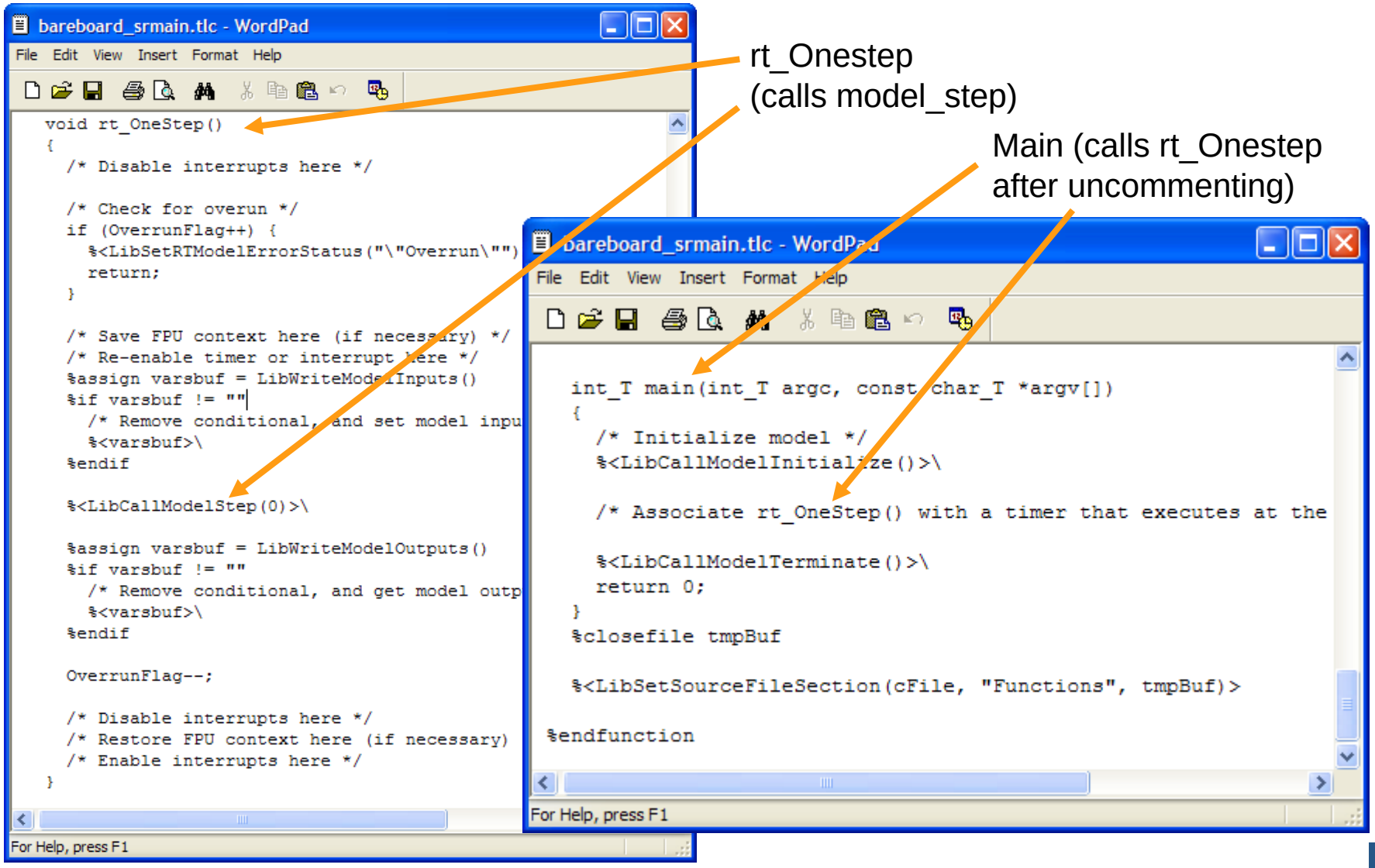
1. **Initialize overrun flags (one for each rate).**
2. **Allocate rate schedule events (one for each rate).**
3. **Check for overrun in the base rate.**
4. **Save FPU context.**
5. **Flag the applicable subrates for the current update.**

Example Scheduling (Multirate, Multitask)



6. Execute base rate task by calling `<model>_step0`.
7. Clear overrun flag of the base rate.
8. Check for overrun in each subrate.
9. Execute each subrate task by calling `<model>_step<i>`.
10. Clear overrun flag of each subrate.
11. Restore FPU context.

Example Template Main Program (Single Rate)



Creating Custom Main Files (Single Rate)

- Copy the example_file_process.tlc to your *targetname/targetname* subdirectory
  - Change the file prefix to use your System Target File
  - Uncomment the ERT_Custom_File_Test assignment
  - Change the bareboard_srmain.tlc to your custom main file
  - Customize further as desired
- Copy the bareboard_srmain.tlc to your *targetname/targetname* subdirectory
  - Change its name prefix to reflect your System Target File
  - Add call to rt_Onestep within the main function
  - Customize further as desired

Student Exercise 2e

Create a Custom Main File

- Copy example_file_process.tlc to htgt_file_process.tlc
  - `cp matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc c:/htgt/htgt/htgt_file_process.tlc`
- Uncomment the ERT_Custom_File_Test assignment (see below)
- Change the bareboard_srmain.tlc to htgt_srmain.tlc (see below)

```
%%  Uncomment this TLC line to execute the example
%%   ||   ||
%%   ||   ||
%%   \/   \/
%assign ERTCustomFileTest = TLC_TRUE

%if EXISTS("ERTCustomFileTest") && ERTCustomFileTest
```

```
%if LibIsSingleRateModel() || LibIsSingleTasking()
  %include "htgt_srmain.tlc"
  %<FcnSingleTaskingMain()>
%else
  %include "bareboard_mrmain.tlc"
  %<FcnMultiTaskingMain()>
%endif
```

Student Exercise 2e (cont.)

Create a Custom Main File

- Copy bareboard_srmain.tlc to htgt_srmain.tlc
  - `cp matlabroot/rtw/c/tlc/mw/bareboard_srmain.tlc c:/htgt/htgt/htgt_srmain.tlc`
- Include a call to rt_OneStep and a printf message to display (see below)

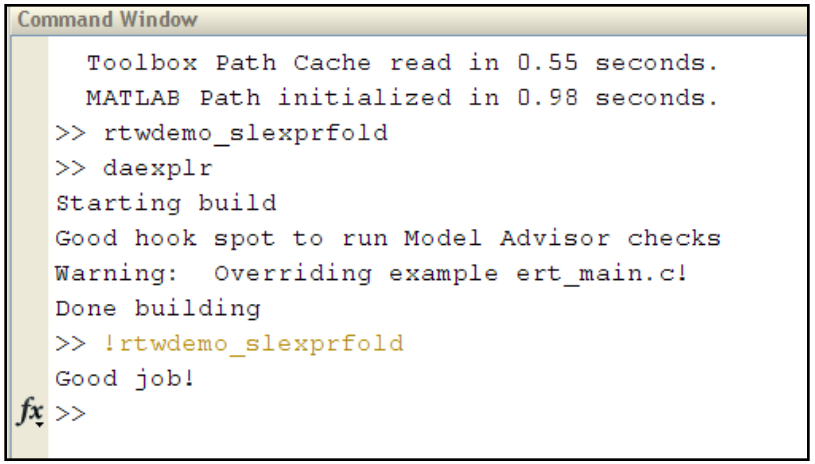
```
int_T main(int_T argc, const char_T *argv[])
{
  /* Initialize model */
  %<LibCallModelInitialize()>\

  rt_OneStep();
  printf("Good job!\n");

  %<LibCallModelTerminate()>\
  return 0;
}
%closefile tmpBuf

%<LibSetSourceFileSection(cFile, "Functions", tmpBuf)>

%endfunction
```

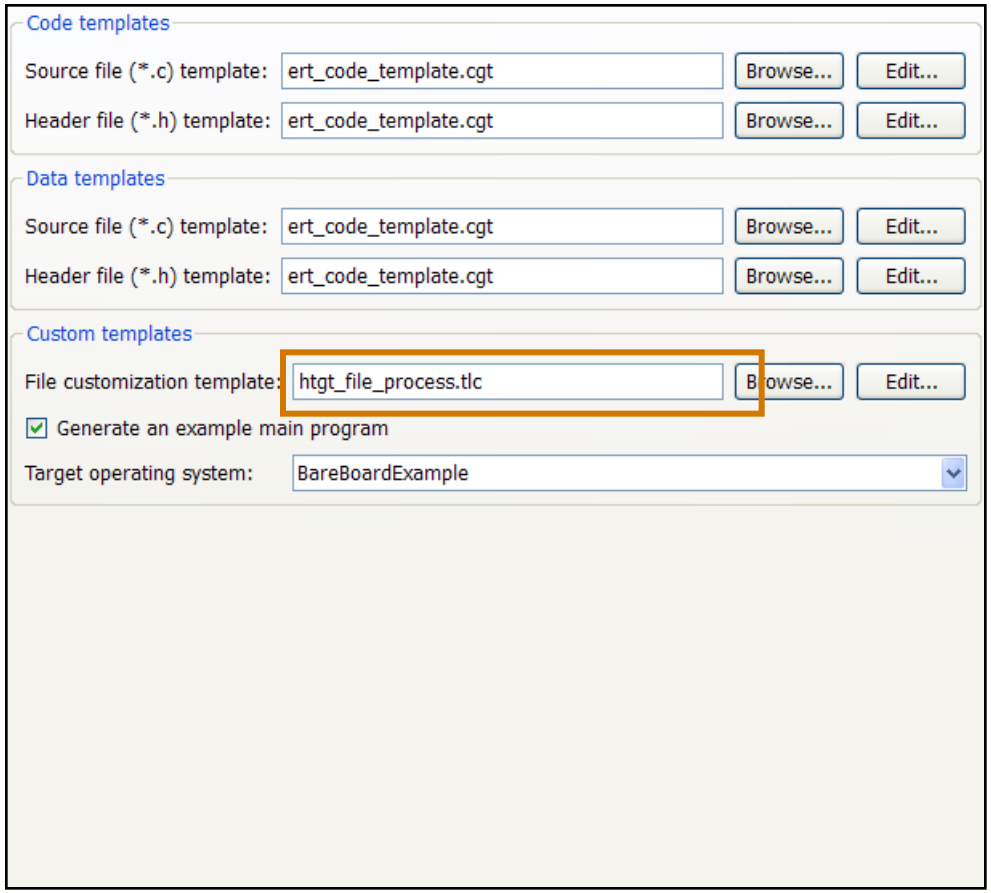
Student Exercise 2e (cont.)

Create a Custom Main File

- Set the file template configuration pane to the htgt as shown below
- Generate and build code for your demo model
- Execute (e.g., >>!mydemo)
- Observe the print message

in the MATLAB window





Creating a Baseline Target

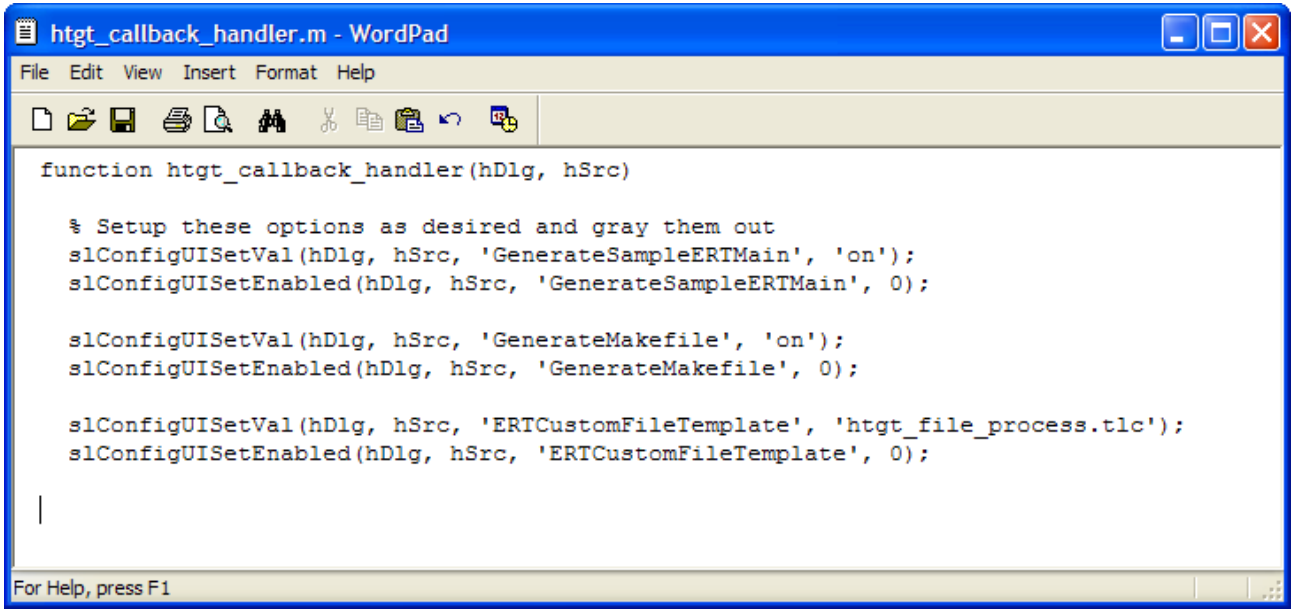
- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Customize Main file
  - **Create Real-Time Workshop call back**
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

Real-Time Workshop Call Back

- Real-Time Workshop call back file can be used with System Target Files to lock down code configuration settings
- Important settings include Hardware Characteristics for target word sizes, make file generation, and main file generation

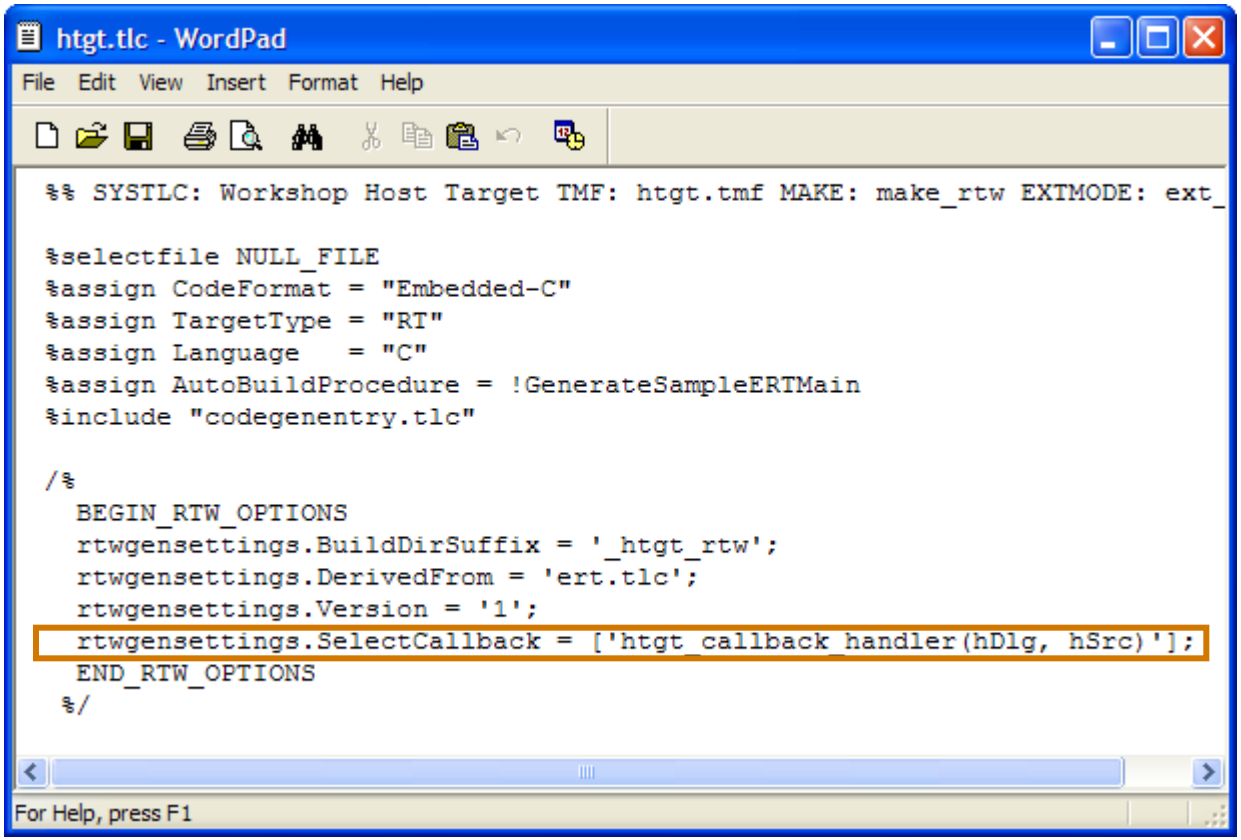
Creating a Call Back Handler

- Create a file in your targetname/targetname subdirectory
- Add a callback handler function for your user interface options



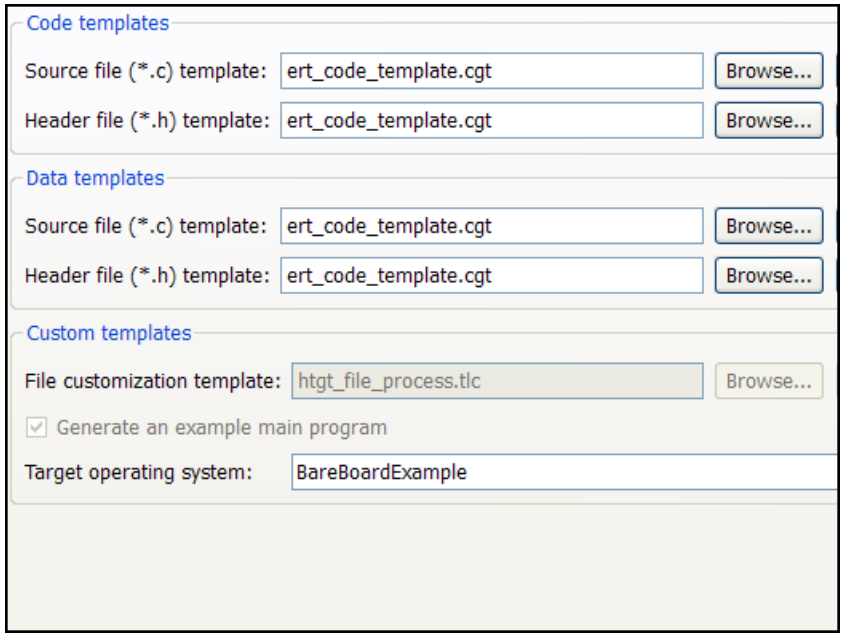
Creating a Call Back Handler (cont.)

- Invoke the call back handler from your system target file



Student Exercise 2f

- Create a simple call back handler as shown in previous slide
- Invoke it from htgt.tlc as shown in previous slide
- Select the htgt system target file
- Observe the configuration pane changes



Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Customize Main file
  - Create Real-Time Workshop call back
  - **Add compiler, chip, and board specific information**
- *A hands-on student exercise*

Add Compiler, chip, and board specific information

- This section is not done for host target since main file customization is sufficient for the example

Creating a Host Target

- Create baseline target
- **Create blockset (if needed)**

Creating Blockset

- **Options for integration legacy code**
- Creating blocks and libraries

Options for integration legacy code

Main options are:

- Hand Crafted S-Functions
- Legacy Code Tool created S-Functions
- Custom Code panels in configuration set

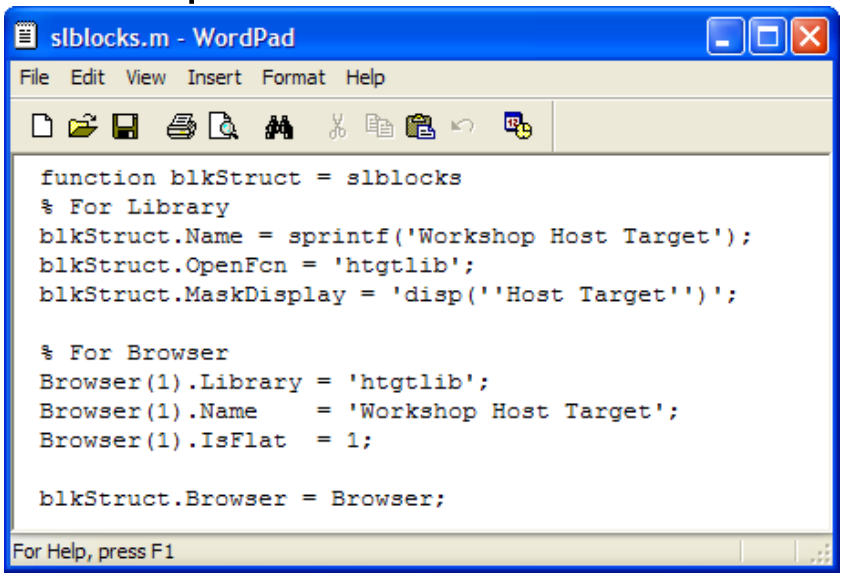
Creating Blockset

- Options for integration legacy code
- **Creating blocks and libraries**

Creating blocks and libraries

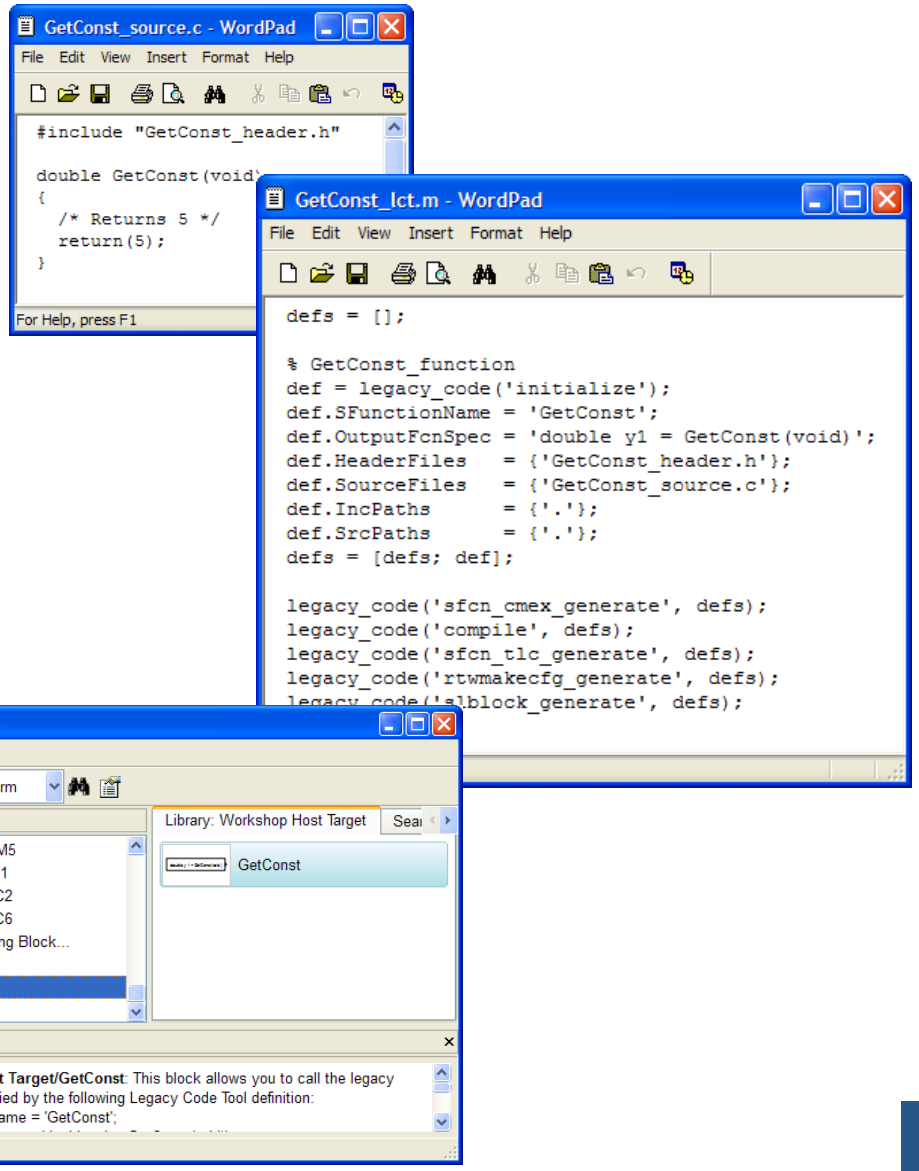
- The process for creating blocks and blocks libraries is well described in documentation so will only be summarized here:
  - Create S-Functions for the device drivers
    - Include user interfaces or masks as needed to pass parameters
  - Create a Library with your device driver blocks
  - In “blocks” directory, c:\htgt\blocks, place:
    - The Library (htgtlib.mdl)
    - S-Functions
    - Driver source code
    - slblocks.m
      - Displays library info

    (as shown to the right)



Student Exercise 2g

- Create an S-Function for GetConst_source.c (and .h)
- Use Legacy Code Tool (see GetConst_lct.m)
- Create htgtlib Library, place S-Function in it
- Build demo with GetConst (see Demo folder)



Agenda

I. Introduction (60 minutes)
II. Creating a Host Target (60 minutes)
III. Lunch
IV. Creating an Embedded Target (60 minutes)
V. Verifying an Embedded Target using PIL test (30 minutes)

Agenda

I. Introduction (60 minutes)
II. Creating a Host Target (60 minutes)
III. Lunch
IV. Creating an Embedded Target (60 minutes)
V. Verifying an Embedded Target using PIL test (30 minutes)

Creating an Embedded Target

- Create baseline
- Create blockset (if needed)

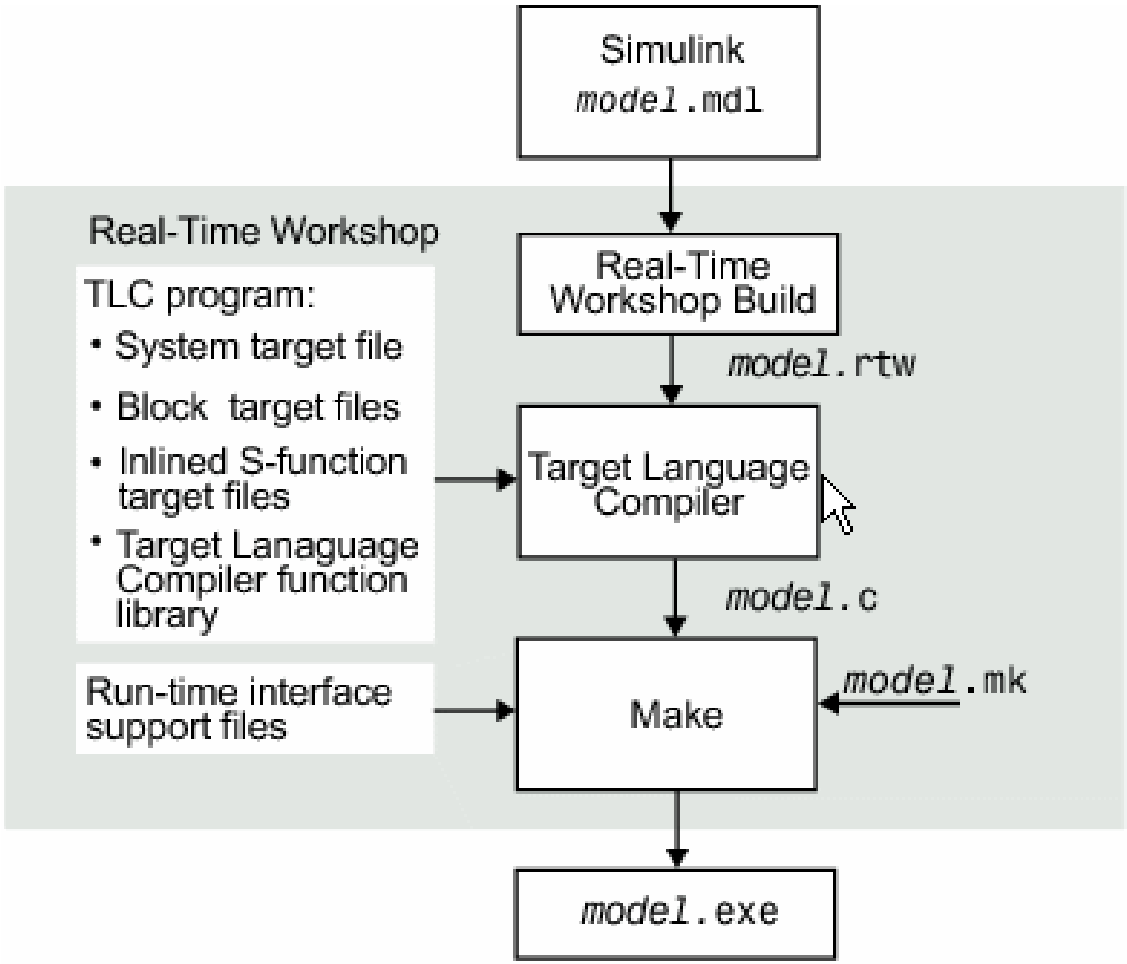
Note: before discussing the baseline process, a brief introduction to Target Language Compiler (TLC) is provided

Target Language Compiler (TLC) Introduction

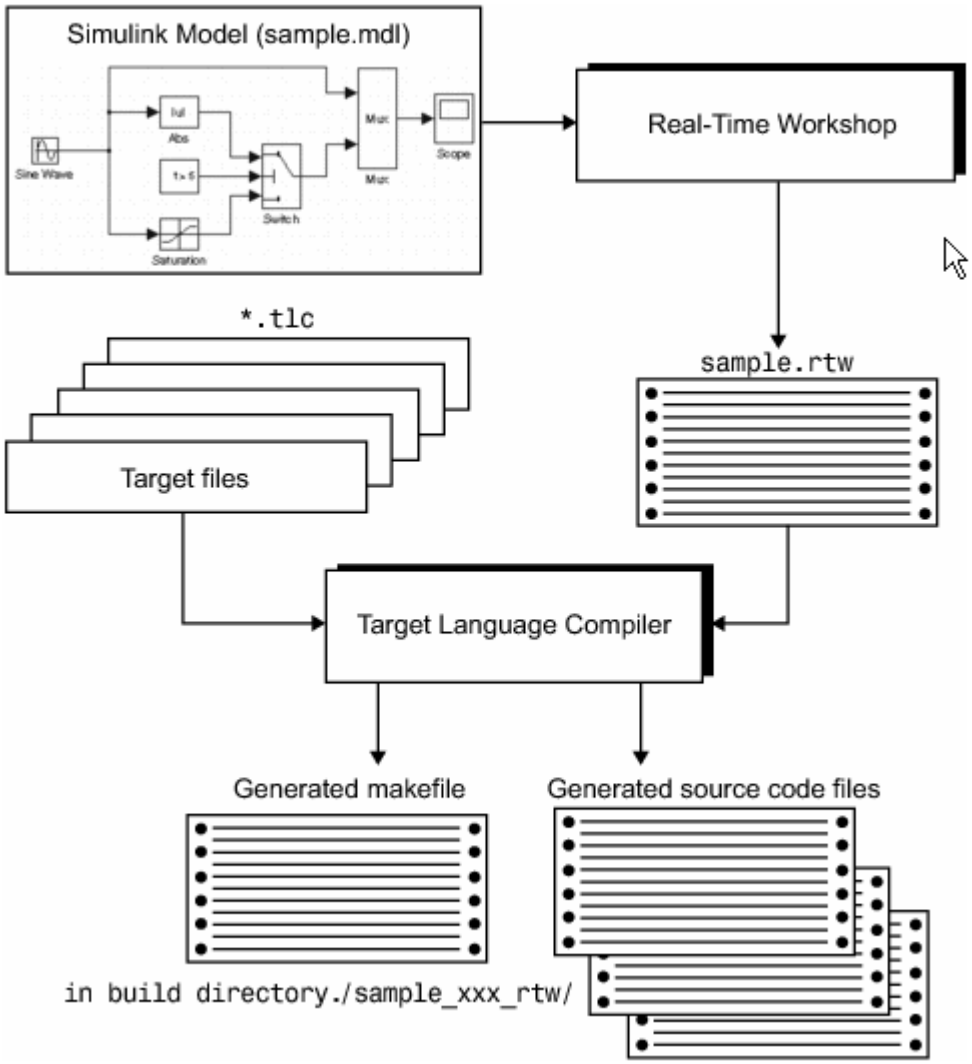
- Powerful for customizing generated code
- Language similar to most scripting languages
- Strong string handling support
- Code generation is controlled by a System Target File written in TLC
  - grt.tlc
  - ert.tlc
  - *custom*.tlc
- Useful for file processing and code file customization

Note in previous releases, TLC was used by expert users for advanced code customization, but newer more powerful APIs have been published to handle these tasks such as Legacy Code Tool and Target Function Libraries.

Target Language Compiler (TLC)



TLC architecture



TLC Language Basics

- The following slides introduce basics of the TLC language
- The material is for background and is not required to complete this workshop or create basic target integration
- Full TLC documentation is provided in the Real-Time Workshop User Guide

TLC program

- A target language file consists of a series of statements of form

  [text | %<expression>]

- %keyword [argument1, argument2, ...]

TLC streams

```
  %openfile  streamId = “filename”
| %openfile  buffer


  %closefile “filename”    %% Closes the file
| %closefile buffer        %% Closes the buffer

   %selectfile   streamId   %% Select file or buffer for output
```

TLC assign

- %assign var = value % declare var and assign its value

  Example:

  %assign var1 = 1

  %assign var2 = “foo”

  %assign sysIdx = sysIdx + 1

  %assign vector = [1, 20, 30,40]

- %<expr> %% Outputs the expression

  Example:

  %<var> %% Outputs the value of var to the current stream

TLC create record

Record – group of data

%createrecord RecName { field fieldVal; field filedVal … }

Example:

%createrecord Rec { Name "Name"; Type "t" }

TLC add to record

%addtorecord Rec1 Rec2 { Name "Name1"; Type "t1" }
%addtorecord Rec1 Rec2 { Name "Name2"; Type "t2" }

```
Rec1 {
  Rec2 {
    Name"Name0"
    Type"t0"
  }
  Rec2 {
    Name"Name1"
    Type"t1"
  }
  Rec2 {
    Name"Name2"
    Type"t2"
  }
    .
    .
}
```

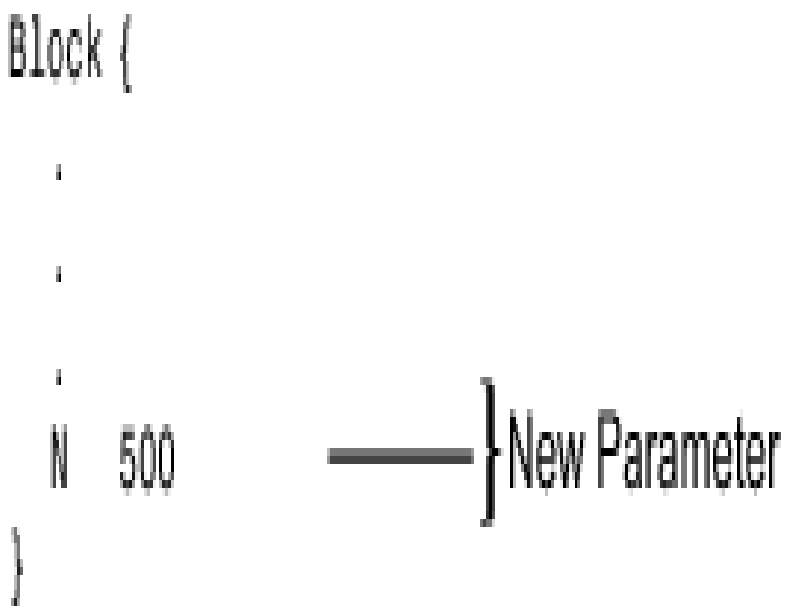
Existing Record

First New Record

Second New Record

TLC add to record

%addtorecord Block N 500 /% Adds N with value 500 to Block %



TLC for loop

```
%foreach idx = length
        %% body
    %endforeach
```

Example:

```
    %foreach  idx = SIZE(arr,1)
        %assign arr(idx) = 1
    %endforeach
```

TLC if statement

```
%if expr
%elseif expr
%elseif expr
%else
%endif
```

Example:

```
%if var1 < 10
    %assign var2 = 1
%elseif var1 == 10
     %assign var2 = 2
%else
     %assign var2 = 3
%endif
```

TLC switch statement

```
%switch expr
%case expr
%break
%case expr
%break
%endswitch
```

Example:

```
%switch val
 %case 0
      %assign out1 = 1
 %break
 %case 1
      %assign out2 = 2
 %break
 %default
       %assign out3 = 3
 %endswitch
```

TLC functions

```
%function func(arg1, arg2, …)  output
        %function body
%endfunction

%function func(arg1,arg2,…) void
    %function body
%endfunction
```

Example:

```
%function  func(arg1, arg2)  output
    %if ISEQUAL(arg1, arg2)
        /* This comment output to the current stream */
        %<arg1> = %<arg2>;
    %else
        %<arg1> = -%<arg2>;
    %endif
%endfunction
```

Calling MATLAB functions

- %matlab

  %matlab disp(2.718)

- %FEVAL

  %assign result = FEVAL(matlab-function-name, rhs1, rhs2, ...

  rhs3, ...);

Creating an Embedded Target

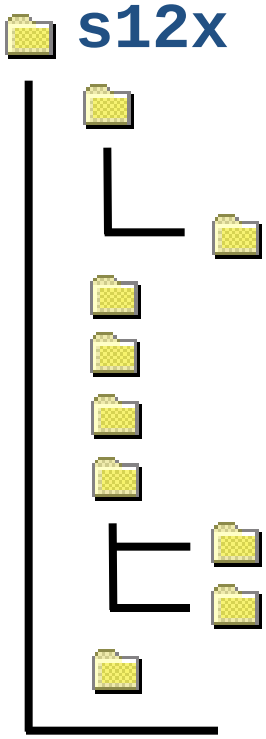
- **Create baseline**
- Create blockset (if needed)

Note: before discussing the baseline process, a brief introduction to Target Language Compiler (TLC) is provided

Creating a Baseline Target

- **Create baseline target**
  - **Develop target directory structure**
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Generate Main file for single and multiple rate models
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

S12x Directory Structure



`blocks` – Source and executable files of I/O driver blocks Library of I/O driver blocks (XML and MATLAB files)

- `tlc_c` – C and TLC files to inline drivers, plus script to build blocks

`cosmic` – Cosmic specific files, interrupt vectors, linker definitions

`demo` – Demo models

`presentation` – Presentation files on installing and running this demo

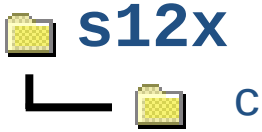
`s12x` – System target file, template makefile, and more

- `+s12xpil` – Custom PIL build class files
- `rtiostream_serial` – Target-side serial driver for PIL testing

`work` – Files generated from simulation and code generation

`run_demo.m` – File to set up demo

Cosmic Subdirectory Structure



`link_def.lkf` – The linker definition file. End users can customize this to change or define different memory sections and other linker information.

`crtsx.s` – The startup file for the target. This is the default supplied by Cosmic. This file is executed when the board is reset or powered up. It initializes the default values for system registers including the stack, then branches to the main function.

`vector_s12x.c` – This file defines the interrupt vector table. When an interrupt occurs, the processor branches to the interrupt specified in the corresponding entry of the interrupt vector table. Note that we load the entire program into flash memory. Thus, the interrupt vector table cannot be modified at run time and must be initialized at compile time.

s12x Subdirectory – Main Control Files

- `s12x.tlc` – System Target File. Contains `rtwoptions` dialog definitions and specifies select callback function, `s12x_callback_handler`. Calls `s12xlib.tlc` before `codegenentry.tlc` to provide definitions needed by `singlerate.tlc` and `multirate.tlc` custom main files.
- `s12x_callback_handler.m` – Specifies default options for the target.
- `s12x_ert.tmf` – Template makefile. Has macros for building libraries involving referenced models and custom files. Includes target `link_def`, which creates `obj_def.inc` and `lib_def.inc` to link the model.
- `s12x_file_process.tlc` – Customizes the generated main for single-rate and multirate builds. Adapted from `example_file_process.tlc` (`matlabroot/toolbox/rtw/ecoder`).
- `s12x_make_rtw_hook.m` – Example hook file. Not used for this target.
- `s12x_srmain.tlc` – Invoked from `s12x_file_process.tlc` to generate main for single rate models. Adapted from `bareboard_srmain.tlc` (`matlabroot/rtw/c/tlc/mw`).
- `s12x_mrmain.tlc` – Invoked from `s12x_file_process.tlc` to generate main for multirate models. Adapted from `bareboard_mrmain.tlc` (`matlabroot/rtw/c/tlc/mw`).
- `s12x_setup.m` – Sets path for the target.
- `s12xlib.tlc` – Common definitions for `s12x_srmain.tlc` and `s12x_mrmain.tlc`.

s12x Subdirectory – Special Files

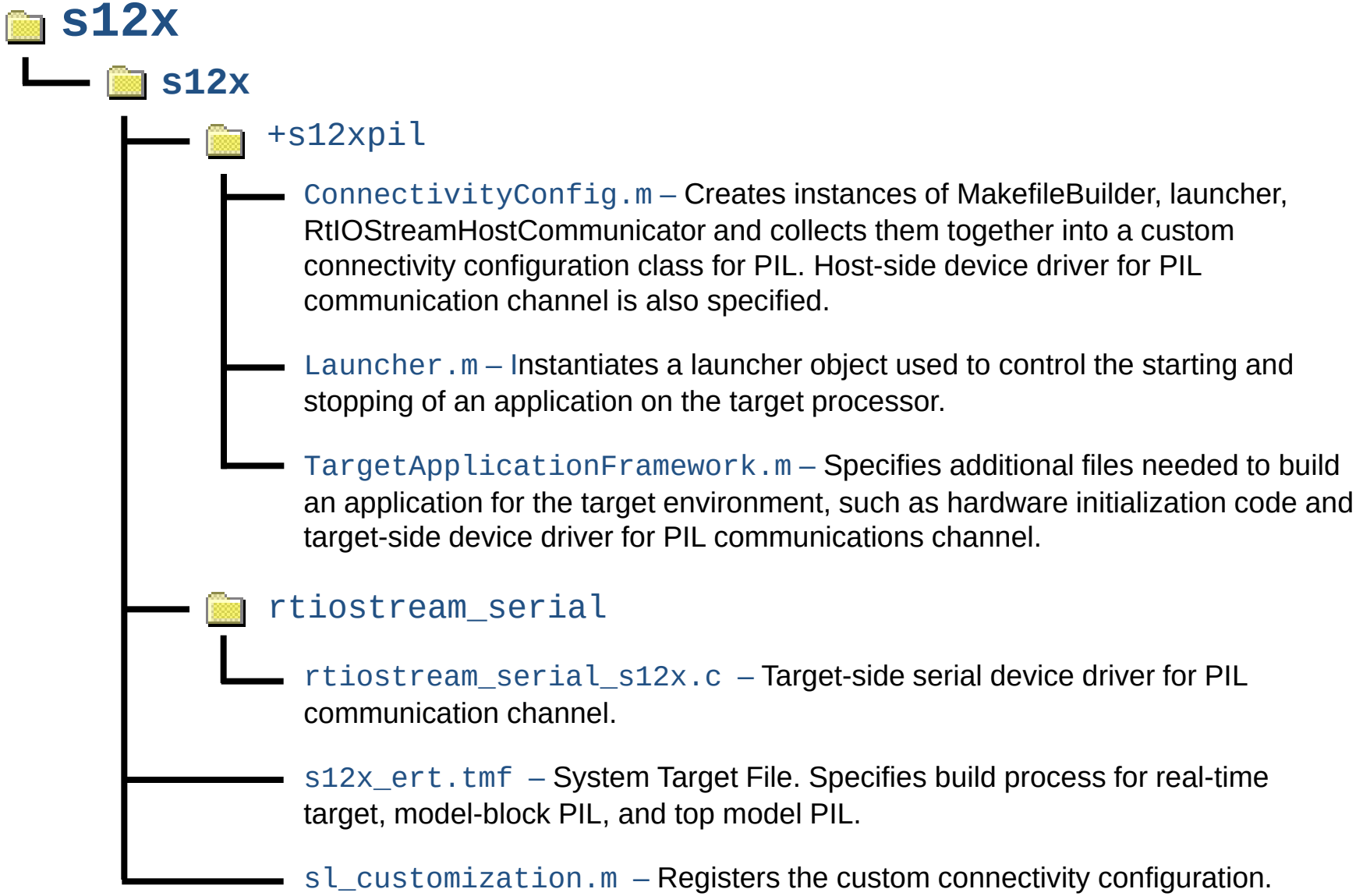

s12x


s12x

- `clock_compute.m` – Computes bit settings for RTICTL register (of clock and reset generator) to set the interrupt time to the step size of the model.
- `generic.pl` – Takes a list and outputs it one line at a time to a specified file. Used by the template makefile to generate `obj_def.inc` and `lib_def.inc` files. The Cosmic linker does not accept list of objects and libraries on the command line, and the Cosmic compiler only accepts the linker definition file. So the template makefile dynamically creates `obj_def.inc` and `lib_def.inc` files using the perl script `generic.pl`.
- `postGenFunc.m` – Specifies location of target files in generated makefile. Invoked before makefile is generated, determines location of target base directory, and sets it in a token `|>S12XROOT<|`. Token is passed back using `buildinfo` and expanded in the generated makefile. To ensure this file is invoked before makefile generation, the following is added to s12x_callback_handler:
  ```
  set_param(bdroot,'PostCodeGenCommand','postGenFunc(buildInfo)');
  ```
- `rtwlib.x12` – Prebuilt MathWorks run-time library. In R2008b and later releases, this file is not needed since all required files are generated dynamically as shared utilities.
- `rtwlib_int.x12` – Prebuilt MathWorks run-time library for use without floating point support. In R2008b, not needed since all required files are generated dynamically as shared utils.

Note: Special files are needed to handle some unique aspects of the Cosmic Cross Development Tools for the S12x target. It is not unusual for embedded targets to require special cases and advanced techniques. Please contact The MathWorks if you find a need for processing not covered in this demo.

s12x Subdirectory – Custom PIL Files



122

Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - **Create System Target file**
  - Customize Makefile
  - Create Real-Time Workshop call back
  - Customize Hook Files for post processing, token additions
  - Generate Main file for single and multiple rate models
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

S12x System Target File

```
%% SYSTLC: Real-Time Workshop Embedded Coder TMF: s12x_ert.tmf \
   MAKE: make_rtw EXTMODE: ext_comm
%%
%% $RCSfile: s12x.tlc,v $
%% $Revision: 9 $
%%
%% Copyright 1994-2004 The MathWorks, Inc.
%% Abstract: Embedded real-time system target file.
%%
%selectfile NULL_FILE

%assign CodeFormat = "Embedded-C"

%assign TargetType = "RT"
%assign Language   = "C"

%assign AutoBuildProcedure = !GenerateSampleERTMain
%include "s12xlib.tlc"
%include "codegenentry.tlc"
```

S12x System Target File

```
%% The contents between 'BEGIN_RTW_OPTIONS' and 'END_RTW_OPTIONS' in this file
%% are used to maintain backward compatibility to R13 and preR13 custom target
%% file only.  If you want to use this file as a template to develop your
%% own system target file, you need to remove the 'CONFIGSET_TARGET_COMPONENT'
%% section at the end of this file.
%%
/%
BEGIN_RTW_OPTIONS
rtwoptions(1).prompt         = 's12x Target Options';
rtwoptions(1).type           = 'Category';
rtwoptions(1).enable         = 'on';
rtwoptions(1).default        = 2;   % number of items under this category
                                    % excluding this one.
rtwoptions(1).popupstrings  = '';
rtwoptions(1).tlcvariable   = '';
rtwoptions(1).tooltip       = '';
rtwoptions(1).callback      = '';
rtwoptions(1).makevariable  = '';

rtwoptions(2).prompt         = 'Board Frequency';
rtwoptions(2).type           = 'Edit';
rtwoptions(2).enable         = 'on';
rtwoptions(2).default        = '4000000';
rtwoptions(2).tlcvariable   = 'S12xBoardFrequency';
rtwoptions(2).tooltip       = 'Enter the clock frequency used by the board.';
rtwoptions(2).callback      = '';
rtwoptions(2).makevariable  = '';
```

S12x System Target File

```
 rtwoptions(3).prompt         = 'Compiler';
 rtwoptions(3).type           = 'Popup';
 rtwoptions(3).popupstrings   = 'cosmic|codewarrior';
 rtwoptions(3).enable         = 'on';
 rtwoptions(3).default        = 'Cosmic';
 rtwoptions(3).tlcvariable    = 'S12xTargetCompiler';
 rtwoptions(3).tooltip        = 'Select the target development compiler.';
 rtwoptions(3).callback       = '';
 rtwoptions(3).makevariable   = 'S12X_TARGET_COMPILER';
 %------------------------------------%
 % Configure RTW code generation settings %
 %------------------------------------%
 rtwgensettings.BuildDirSuffix = '_s12x_rtw';
 rtwgensettings.DerivedFrom = 'ert.tlc';
 rtwgensettings.Version = '1';
 rtwgensettings.SelectCallback = ['s12x_callback_handler(hDlg, hSrc)'];
 END_RTW_OPTIONS
%/
```

Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - **Customize Makefile**
  - Create Real-Time Workshop call back
  - Customize Hook Files for post processing, token additions
  - Generate Main file for single and multiple rate models
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

S12x Template makefile

```
MATLAB_ROOT             = |>MATLAB_ROOT<|
ALT_MATLAB_ROOT         = |>ALT_MATLAB_ROOT<|
MATLAB_BIN              = |>MATLAB_BIN<|
ALT_MATLAB_BIN          = |>ALT_MATLAB_BIN<|
S_FUNCTIONS             = |>S_FUNCTIONS<|
S_FUNCTIONS_LIB         = |>S_FUNCTIONS_LIB<|
NUMST                   = |>NUMST<|
NCSTATES                = |>NCSTATES<|
BUILDARGS               = |>BUILDARGS<|
MULTITASKING            = |>MULTITASKING<|
INTEGER_CODE            = |>INTEGER_CODE<|
MAT_FILE                = |>MAT_FILE<|
ONESTEPFCN              = |>COMBINE_OUTPUT_UPDATE_FCNS<|
TERMFCN                 = |>INCLUDE_MDL_TERMINATE_FCN<|
B_ERTSFCN               = |>GENERATE_ERT_S_FUNCTION<|
MEXEXT                  = |>MEXEXT<|
EXT_MODE                = |>EXT_MODE<|
TMW_EXTMODE_TESTING     = |>TMW_EXTMODE_TESTING<|
EXTMODE_TRANSPORT       = |>EXTMODE_TRANSPORT<|
EXTMODE_STATIC          = |>EXTMODE_STATIC_ALLOC<|
EXTMODE_STATIC_SIZE     = |>EXTMODE_STATIC_ALLOC_SIZE<|
MULTI_INSTANCE_CODE     = |>MULTI_INSTANCE_CODE<|
MODELREFS                 = |>MODELREFS<|
SHARED_SRC                = |>SHARED_SRC<|
SHARED_SRC_DIR            = |>SHARED_SRC_DIR<|
SHARED_BIN_DIR            = |>SHARED_BIN_DIR<|
SHARED_LIB                = |>SHARED_LIB<|
GEN_SAMPLE_MAIN           = |>GEN_SAMPLE_MAIN<|
MEX_OPT_FILE              = |>MEX_OPT_FILE<|
TARGET_LANG_EXT           = |>TARGET_LANG_EXT<|
MEX_OPT_FILE              = |>MEX_OPT_FILE<|
PORTABLE_WORDSIZES        = |>PORTABLE_WORDSIZES<|
SHRLIBTARGET              = |>SHRLIBTARGET<|
S12X_TARGET_COMPILER      = |>S12X_TARGET_COMPILER<|
S12XROOT                  = |>S12XROOT<|
```

S12x Template makefile

```
#---------------------------- Include Path ---------------------------------

MATLAB_INCLUDES = \
 -i $(MATLAB_ROOT)\rtw\c\ert \
 -i $(MATLAB_ROOT)\extern\include \
 -i $(MATLAB_ROOT)\simulink\include \
 -i $(MATLAB_ROOT)\rtw\c\src \
 -i $(MATLAB_ROOT)\rtw\c\src\ext_mode\common

# Additional includes
# Additional includes

ADD_INCLUDES = \
|>START_EXPAND_INCLUDES<|   -i|>EXPAND_DIR_NAME<| \
|>END_EXPAND_INCLUDES<|


SHARED_INCLUDES =
ifneq ($(SHARED_SRC_DIR),)
SHARED_INCLUDES = -i$(SHARED_SRC_DIR)
endif

INCLUDES = -i. -i$(RELATIVE_PATH_TO_ANCHOR) $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
           $(COMPILER_INCLUDES) $(USER_INCLUDES) $(MODELREF_INC_PATH) \
           $(SHARED_INCLUDES)

INCLUDES += -i$(MATLAB_ROOT)\rtw\c\ert
```

S12x Template makefile

```
#---------------------------------- C Flags ------------------------------------

# Optimization Options
OPT_OPTS = $(DEFAULT_OPT_OPTS)

# General User Options
OPTS =

# Compiler options, etc:
CC_OPTS = $(OPT_OPTS) $(OPTS) $(ANSI_OPTS) $(EXT_CC_OPTS)

CPP_REQ_DEFINES = -dMODEL=$(MODEL) -dNUMST=$(NUMST) -dNCSTATES=$(NCSTATES) \
           -dMT=$(MULTITASKING) \
           -dMAT_FILE=$(MAT_FILE) -dINTEGER_CODE=$(INTEGER_CODE) \
           -dONESTEPFCN=$(ONESTEPFCN) -dTERMFCN=$(TERMFCN) \
           -dHAVESTDIO -dMULTI_INSTANCE_CODE=$(MULTI_INSTANCE_CODE)\
           -dADD_MDL_NAME_TO_GLOBALS=$(ADD_MDL_NAME_TO_GLOBALS)

ifeq ($(MODELREF_TARGET_TYPE),SIM)
CPP_REQ_DEFINES += -dMDL_REF_SIM_TGT=1
else
CPP_REQ_DEFINES += -dMT=$(MULTITASKING)
endif

# -no turns the optimizer off
HC12FLGS_OPT = -no

# +debug produces debug information in the code
HC12FLGS_DEBUG = +debug

HC12FLGS=-p-nwx -o -c $(HC12FLGS_DEBUG) -l   +xe $(HC12FLGS_OPT) -cl ./ -co./ -tc:\temp -dF_FAR="" \
   -dV_FAR="" -dG_CAL="@gpage"  -dINLINE_FUNCTION=""

# HC12FLGS=-p-nwx -o -c +debug -l   +xe -no -cl ./ -co./ -tc:\temp -dF_FAR="" \
#   -dV_FAR="" -dG_CAL="@gpage"  -dINLINE_FUNCTION=""

CFLAGS_RAW = $(HC12FLGS) $(CC_OPTS) $(CPP_REQ_DEFINES) $(INCLUDES)
CFLAGS = $(subst -D,-d,$(CFLAGS_RAW))
```

S12x Template makefile

```
#------------------------- Additional Libraries -----------------------------
LIBS =
|>START_PRECOMP_LIBRARIES<|
ifeq ($(OPT_OPTS),$(DEFAULT_OPT_OPTS))
  ifeq ($(INTEGER_CODE),0)
     ifeq ($(MODELREF_TARGET_TYPE),SIM)
        LIBS += |>EXPAND_LIBRARY_LOCATION<|\|>EXPAND_LIBRARY_NAME<|_rtwsfcn.lib
     else
        LIBS += |>EXPAND_LIBRARY_LOCATION<|\|>EXPAND_LIBRARY_NAME<|.x12
     endif
   else
      LIBS += |>EXPAND_LIBRARY_NAME<|_int.x12
   endif
else
  LIBS += |>EXPAND_LIBRARY_NAME<|.x12
endif
|>END_PRECOMP_LIBRARIES<| |>START_EXPAND_LIBRARIES<|
LIBS += |>EXPAND_LIBRARY_NAME<|.x12 |>END_EXPAND_LIBRARIES<|
LIBS += $(S_FUNCTIONS_LIB) $(INSTRUMENT_LIBS)

ifeq ($(SFCN),1)
LIBFIXPT = $(MATLAB_ROOT)\extern\lib\win32\lcc\libfixedpoint.lib
LIBS += $(LIBFIXPT)
endif
```

S12x Template makefile – Source Files

```
#---------------------------- Source Files -----------------------------------

ifeq ($(SFCN),0)
    SRCS   = $(MODULES)  $(S_FUNCTIONS)
    SRC_DEP =
    ifeq ($(MODELREF_TARGET_TYPE), NONE)
      ifeq ($(MAKEFILEBUILDER_TGT), 1)
    #--- Stand-alone model for PIL (.x12) ---
        PRODUCT             = $(MODEL).x12
        BUILD_PRODUCT_TYPE = "executable"
        SRCS                += $(EXT_SRC)
        BIN_SETTING         = $(LD) $(LDFLAGS) $(ADDITIONAL_LDFLAGS) -o $(PRODUCT) $(SYSLIBS)
      else
    #--- Stand-alone model (.exe) ---
        PRODUCT             = $(RELATIVE_PATH_TO_ANCHOR)/$(MODEL).x12
        BUILD_PRODUCT_TYPE = "executable"
        SRCS                += ert_main.c $(MODEL).c  $(EXT_SRC)
      endif
    else
    #--- sub-model for RTW ---
      PRODUCT              = $(MODELLIB)
      BUILD_PRODUCT_TYPE = library
    endif
endif

USER_SRCS =
USER_OBJS        = $(USER_SRCS:.c=.o)
LOCAL_USER_OBJS = $(notdir $(USER_OBJS))

S12X_COBJS = vector_s12x.o timhndlr.o portpinterrupt.o s12x_sci.o
PIL_TGT_OBJS = rtiostream_serial_s12x.o
AS_OBJS = crtsx.o

# For regular builds
OBJS      = $(SRCS:.c=.o) $(AS_OBJS)  $(USER_OBJS) $(S12X_COBJS)
# For PIL cosimulation in R2009a, vector_s12x.o is not required
PIL_OBJS  = $(SRCS:.c=.o) $(AS_OBJS)  $(USER_OBJS) timhndlr.o portpinterrupt.o s12x_sci.o
LINK_OBJS = $(SRCS:.c=.o) $(AS_OBJS)  $(LOCAL_USER_OBJS)
SHARED_OBJS := $(addsuffix .o, $(basename $(wildcard $(SHARED_SRC))))
```

S12x Template makefile - Rules

```
#---------------------------------- Rules ---------------------------------------

ifeq ($(MODELREF_TARGET_TYPE),NONE)
  ifeq ($(MAKEFILEBUILDER_TGT), 1)
#--- Stand-alone model for PIL (.x12) ---
   $(PRODUCT) : $(PIL_OBJS) $(MODELLIB) $(SHARED_LIB) $(LIBS) $(SRC_DEP) $(PIL_TGT_OBJS) link_def
    @cmd /C "echo ### Linking ..."
    $(LINKCMD)  -o $(MODEL).x12 -m $(MODEL).map link_def.lkf
    clabs  $(MODEL).x12
    cvdwarf  $(MODEL).x12
    chex     -s -o $(MODEL).s19  $(MODEL).x12

    @echo ### Created $(BUILD_PRODUCT_TYPE): $@
  else
#--- Stand-alone model (.exe) ---
   $(PRODUCT) : $(OBJS) $(SHARED_LIB) $(LIBS) $(SRC_DEP) link_def
    @cmd /C "echo ### Linking ..."
    $(LINKCMD)  -o $(MODEL).x12 -m $(MODEL).map  link_def.lkf
    clabs  $(MODEL).x12
    cvdwarf  $(MODEL).x12
    chex     -s -o $(MODEL).s19  $(MODEL).x12

    @echo ### Created $(BUILD_PRODUCT_TYPE): $@
  endif
else
#--- For future work ---
 ifeq ($(MODELREF_TARGET_TYPE),SIM)
    @echo ### This function has not been implemented yet...
 else
#--- For referenced model, create a library ---
  $(PRODUCT) : $(OBJS) $(SHARED_LIB) $(LIBS) $(SRC_DEP)
    $(LIBCMD) -v -c $@ $(LINK_OBJS)
    @echo ### Created $(MODELLIB)
    @echo ### Created $(BUILD_PRODUCT_TYPE): $@
  endif
endif
```

S12x Template makefile – Rules (cont)

```
%.o : $(S12XROOT)/$(S12X_TARGET_COMPILER)/%.s
     @echo Building assembler file $@ to $<
     $(AS)   -o $@   $<

%.o : $(S12XROOT)/blocks/tlc_c/%.c
     @echo Using S12XROOT/block/tlc_c target for $<
     $(CC)   $(CFLAGS) -o $@   $<

%.o : $(S12XROOT)/$(S12X_TARGET_COMPILER)/%.c
     $(CC)   $(CFLAGS) -o $@   $<

%.o : $(S12XROOT)/%.c
     $(CC)   $(CFLAGS) -o $@   $<

%.o : $(RELATIVE_PATH_TO_ANCHOR)/%.c
     $(CC)    $(CFLAGS) $<

%.o : $(MATLAB_ROOT)/rtw/c/ert/%.c
     $(CC)    $(CFLAGS) $<

%.o : $(MATLAB_ROOT)/rtw/c/src/%.c
     $(CC)    $(CFLAGS) $<

%.o : $(MATLAB_ROOT)/rtw/c/src/ext_mode/common/%.c
     $(CC)    $(CFLAGS) $<

%.o : $(MATLAB_ROOT)/rtw/c/src/ext_mode/tcpip/%.c
     $(CC)    $(CFLAGS) $<

%.o : $(MATLAB_ROOT)/rtw/c/src/ext_mode/serial/%.c
     $(CC)    $(CFLAGS) $<

%.o : $(MATLAB_ROOT)/rtw/c/src/ext_mode/custom/%.c
     $(CC)    $(CFLAGS) $<

%.o : $(MATLAB_ROOT)/rtw/c/src/rtiostream/rtiostreamtcpip/%.c
     $(CC) $(CFLAGS) "$<"
```

S12x Template makefile – Rules(cont)

```
%.o : $(S12XROOT)/rtiostream_serial/%.c
    @echo Using S12XROOT/rtiostream_serial target for $<
    $(CC)  $(CFLAGS) -o $@  $<

# For finding reference model object file, like "pil_component.o"
%.o : ../%.c
    $(CC)  $(CFLAGS) -o $@  $<

|>START_EXPAND_RULES<|%.o : |>EXPAND_DIR_NAME<|/%.c
    $(CC)   $(CFLAGS) $<

|>END_EXPAND_RULES<|

%.o : $(MATLAB_ROOT)/simulink/src/%.c
    $(CC)  -Fo$(@F) $(CFLAGS) $<

%.o : $(SHARED_SRC_DIR)/%.c
    @echo Using SHARED_SRC_DIR for $<
    $(CC)  $(CFLAGS) -o $@ -co$(SHARED_SRC_DIR) $<

%.o : %.c
    $(CC)  $(CFLAGS) $<
```

S12x Template makefile - libraries

```
# ---------------------------------Libraries---------------------------------

|>START_EXPAND_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<|     |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|


|>EXPAND_LIBRARY_NAME<|.x12 : $(MAKEFILE) rtw_proj.tmw $(MODULES_|>EXPAND_LIBRARY_NAME<|)
    @echo ### Creating $@
    @if exist $@ del $@
    $(LIBCMD) -v -c $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|)
    @echo ### $@ Created

|>END_EXPAND_LIBRARIES<|

|>START_PRECOMP_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<|     |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.x12 : $(MAKEFILE) rtw_proj.tmw $(MODULES_|>EXPAND_LIBRARY_NAME<|)
    @echo ### Creating $@
    @if exist $@ del $@
    $(LIBCMD) -v -c $@  $(MODULES_|>EXPAND_LIBRARY_NAME<|)
    @echo ### $@ Created
|>END_PRECOMP_LIBRARIES<|

|>START_PRECOMP_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<|_int.x12 = \
|>START_EXPAND_MODULES<|     |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|_int.x12 : $(MAKEFILE) rtw_proj.tmw $(MODULES_|>EXPAND_LIBRARY_NAME<|)
    @echo ### Creating $@
    @if exist $@ del $@
    $(LIBCMD) -v -c $@  $(MODULES_|>EXPAND_LIBRARY_NAME<|)
    @echo ### $@ Created
|>END_PRECOMP_LIBRARIES<|
```

Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - **Customize Hook Files for post processing, token additions**
  - Generate Main file for single and multiple rate models
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

S12x Make Hook File

- Make Hook was created but not used for S12x

s12x_make_rtw_hook.m - WordPad

File Edit View Insert Format Help

```
function s12x_make_rtw_hook(method, modelName, rtwRoot, tmf, buildOpts, buildArgs, rt
% Make RTW hook file for hcs12 target.

% Copyright 2002 The MathWorks, Inc.
% $Revision: 9 $
% $Date: 2008-09-18 16:56:27 -0400 (Thu, 18 Sep 2008) $

fprintf(1, '\n### %s: %s (%s)\n', mfilename, method, modelName);

switch method
    case 'entry'
        fprintf(1,['\n### Starting Real-Time Workshop build procedure for ',...
            'model: %s\n'],modelName);
    case 'before_tlc'
       % No action

    case 'after_tlc'
        % No action

    case 'before_make'
        % Generate files containing lists for building any necessary libraries.
        % For example, rtwlib.lib.
        fprintf(1,['\n### Real Time Workshop before make\n']);

    case 'after_make'
         fprintf(1,['\n### Real Time Workshop after make\n']);

    case 'exit'
        % Get the action
```

For Help, press F1

Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - **Generate Main file for single and multiple rate models**
  - Create Real-Time Workshop call back
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

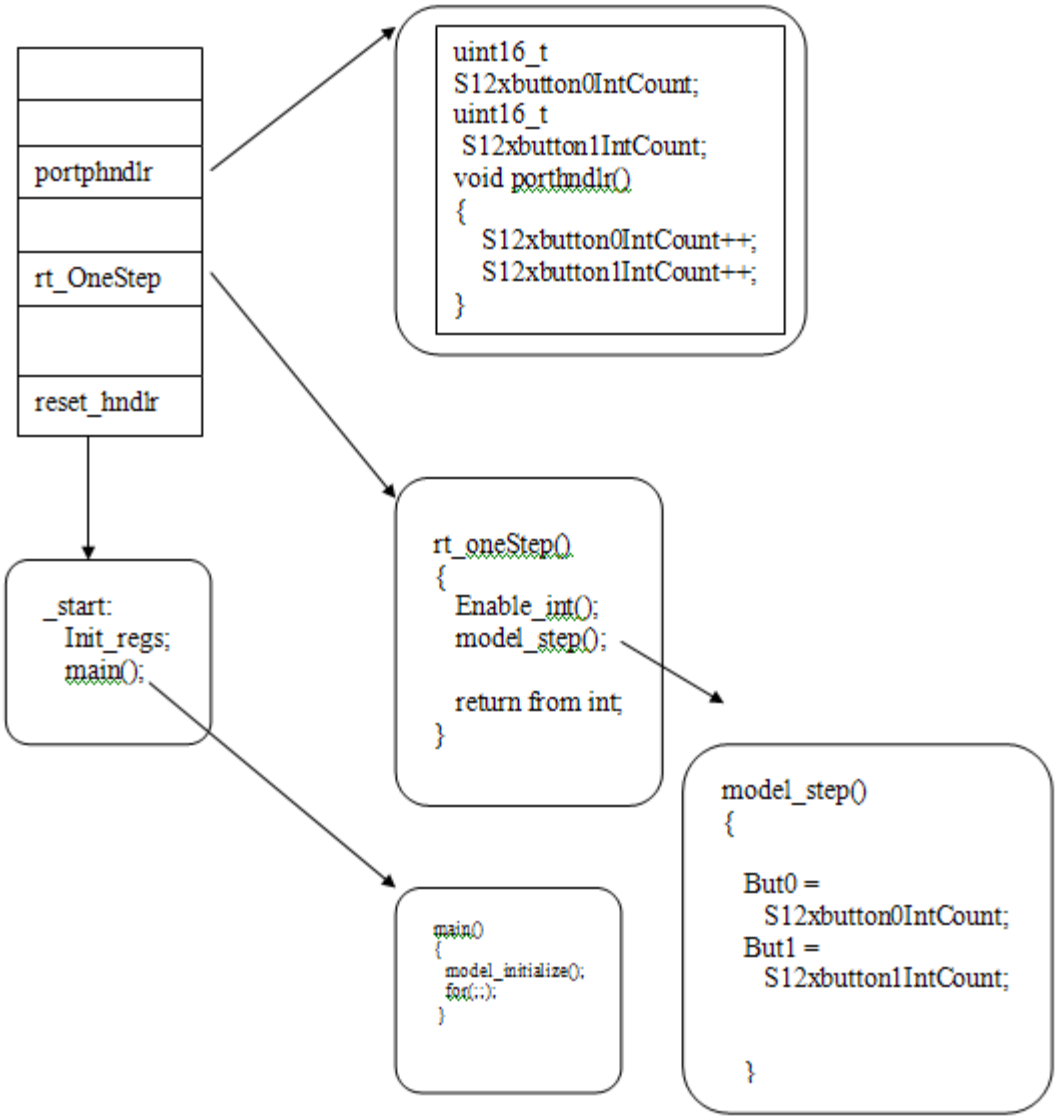
S12x File Processing Hook

```
%%  Uncomment this TLC line to execute the example
%%   ||   ||
%%   ||   ||
%%   \/   \/

%if !IsModelReferenceTarget()
  %assign ERTCustomFileTest = TLC_TRUE

  %assign CompiledModel.GenerateSampleERTMain = TLC_FALSE

  %if EXISTS("ERTCustomFileTest") && ERTCustomFileTest == TLC_TRUE

    %% Need to set the template compliance flag before you can use the API
    %<LibSetCodeTemplateComplianceLevel(1)>

    %if LibIsSingleRateModel() || LibIsSingleTasking()
       %include "s12x_srmain.tlc"
       %<s12xSingleTaskingMain()>
    %else
       %include "s12x_mrmain.tlc"
       %<s12xFcnMultiTaskingMain()>
    %endif
  %endif
%endif
%<TgtLibS12XCreateS12XMcuHeader()>
```

S12x Template Main File (Single Rate)

```
  int_T main(int_T argc, const char_T *argv[])
  {
    /* Initialize model */

    %<LibCallModelInitialize()>

    _asm("cli");
    configureRti();


    %if S12xBuildForSimulator == 0
    for (;;);
    %else
    for (;;) {
       /*
        * On the simulator, just call rt_OneStep for each step of
        * the simulation. Interrupts cannot be simulated.
        */
       rt_OneStep();
    }
    %endif

    %<LibCallModelTerminate()>
    return 0;
  }
  %closefile tmpBuf

  %<LibSetSourceFileSection(cFile, "Functions", tmpBuf)>

%endfunction
```

Detailed S12X Scheduling



Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Generate Main file for single and multiple rate models
  - **Create Real-Time Workshop call back**
  - Add compiler, chip, and board specific information
- *A hands-on student exercise*

S12x Call Back

```
function s12x_callback_handler(hDlg, hSrc)
  set_param(bdroot,'PostCodeGenCommand','postGenFunc(buildInfo)');
  curVal=slConfigUIGetVal(hDlg, hSrc, 'ModelReferenceCompliant');
  slConfigUISetVal(hDlg, hSrc, 'ModelReferenceCompliant', 'on');
  curVal=slConfigUIGetVal(hDlg, hSrc, 'ModelReferenceCompliant');
  slConfigUISetEnabled(hDlg, hSrc, 'ModelReferenceCompliant', false);

  slConfigUISetVal(hDlg, hSrc, 'CompOptLevelCompliant', 'on');
  slConfigUISetEnabled(hDlg, hSrc, 'CompOptLevelCompliant', false);

  % Setup these options as desired and gray them out
  slConfigUISetVal(hDlg, hSrc, 'ProdHWDeviceType', 'Motorola HC12');
  slConfigUISetEnabled(hDlg, hSrc, 'ProdHWDeviceType', 0);
  slConfigUISetVal(hDlg, hSrc, 'ProdEqTarget','on');
  slConfigUISetEnabled(hDlg, hSrc, 'ProdEqTarget', 0);

  slConfigUISetVal(hDlg, hSrc, 'ZeroExternalMemoryAtStartup','off');
  slConfigUISetVal(hDlg, hSrc, 'ZeroInternalMemoryAtStartup','off');

  slConfigUISetVal(hDlg, hSrc, 'NoFixptDivByZeroProtection', 'on');
  slConfigUISetVal(hDlg, hSrc, 'EfficientFloat2IntCast', 'on');

  slConfigUISetVal(hDlg, hSrc, 'GenerateSampleERTMain', 'on');
  slConfigUISetEnabled(hDlg, hSrc, 'GenerateSampleERTMain', 0);

  slConfigUISetVal(hDlg, hSrc, 'GenerateMakefile', 'on');
  slConfigUISetEnabled(hDlg, hSrc, 'GenerateMakefile', 0);

  slConfigUISetVal(hDlg, hSrc, 'GenerateErtSFunction', 'off');
  slConfigUISetEnabled(hDlg, hSrc, 'GenerateErtSFunction', 0);

  slConfigUISetVal(hDlg, hSrc, 'MultiInstanceERTCode', 'off');
  slConfigUISetEnabled(hDlg, hSrc, 'MultiInstanceERTCode', 0);
```

S12x Call Back (cont.)

```
slConfigUISetVal(hDlg, hSrc, 'ERTCustomFileTemplate', 's12x_file_process.tlc');
slConfigUISetEnabled(hDlg, hSrc, 'ERTCustomFileTemplate', 0);

slConfigUISetVal(hDlg, hSrc, 'SupportNonInlinedSFcns', 'off');
%slConfigUISetEnabled(hDlg, hSrc, 'SupportNonInlinedSFcns', 0);

slConfigUISetVal(hDlg, hSrc, 'UtilityFuncGeneration', 'Auto');
%slConfigUISetEnabled(hDlg, hSrc, 'UtilityFuncGeneration', 0);

slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 0);
slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', 0);

slConfigUISetVal(hDlg, hSrc, 'InitFltsAndDblsToZero','off');

slConfigUISetVal(hDlg, hSrc, 'ZeroExternalMemoryAtStartup','off');

slConfigUISetVal(hDlg, hSrc, 'ZeroInternalMemoryAtStartup','off');

slConfigUISetVal(hDlg, hSrc, 'PurelyIntegerCode','off');

slConfigUISetVal(hDlg, hSrc, 'SupportNonFinite','off');

slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn','off');

slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn',0);

slConfigUISetVal(hDlg, hSrc, 'SupportNonInlinedSFcns','off');

slConfigUISetVal(hDlg, hSrc, 'ERTFirstTimeCompliant','on');

slConfigUISetVal(hDlg, hSrc, 'TargetLibSuffix', '.x12');
slConfigUISetEnabled(hDlg, hSrc, 'TargetLibSuffix', 0);

% Set the precompilation directory to the s12x directory.
fpath = which(mfilename());
[s12xdir, filename] = fileparts(fpath);
set_param(bdroot,'TargetPreCompLibLocation',s12xdir);
```

Creating a Baseline Target

- Create baseline target process
  - Develop target directory structure
  - Create System Target file
  - Customize Makefile
  - Customize Hook Files for post processing, token additions
  - Generate Main file for single and multiple rate models
  - Create Real-Time Workshop call back
  - **Add compiler, chip, and board specific information**
- *A hands-on student exercise*

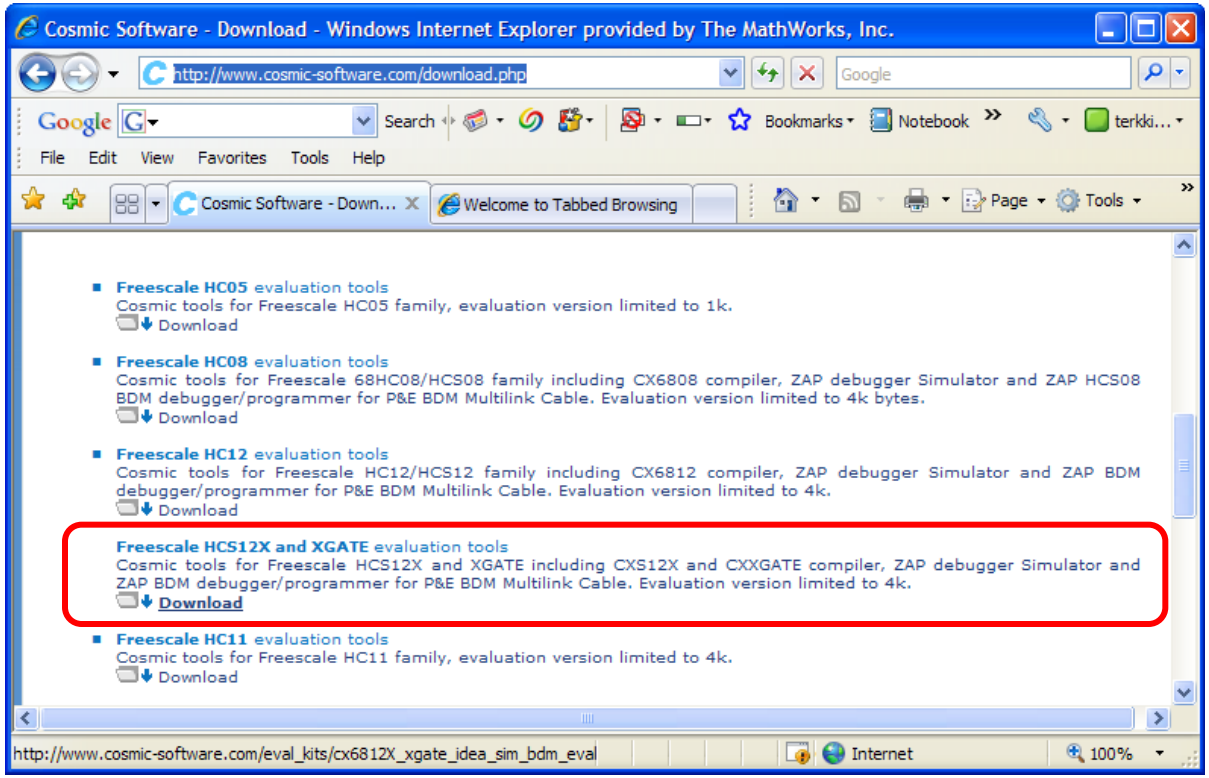
S12x ISR

- Must provide a time based interrupt service routine (ISR) to meet the following requirements
  - Clear the Timebase Interrupt flag to acknowledge interrupt
  - Indirectly call model_step via a rt_OneStep function

```
/*     INTERRUPT VECTORS TABLE S12X
 *     Copyright (c) 2004 by COSMIC Software
 */
@interrupt @near void _stext(void); /* startup routine */

/*     dummy interrupt routine
 */
static @interrupt @near void dummit(void)
       {
       }


extern @interrupt @near void rt_OneStep(void);
```

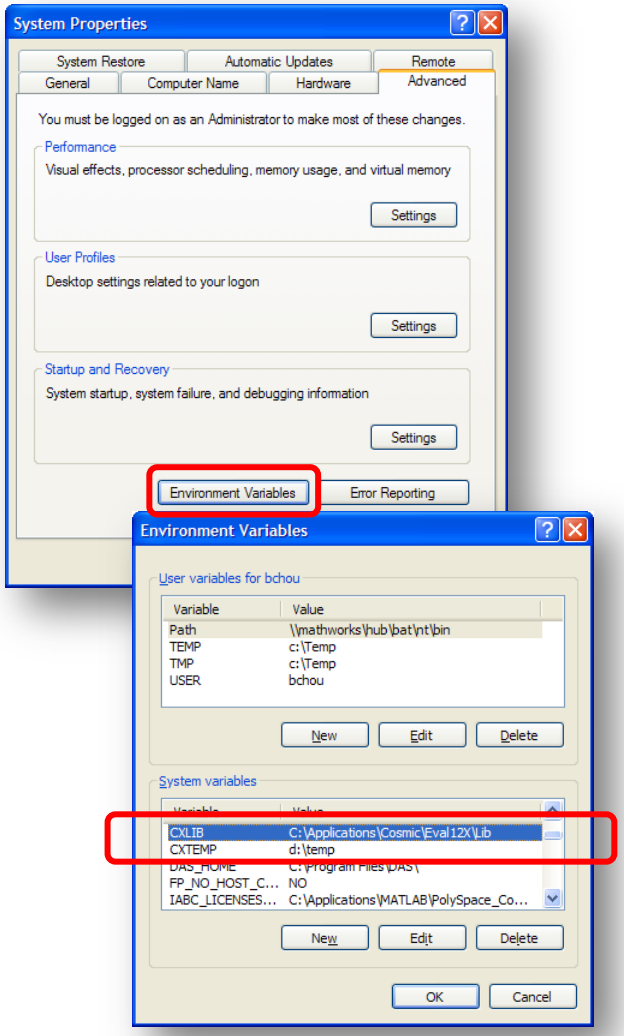
Class Exercise #3

- Download Freescale HCS12X and XGATE evaluation tools
  - CX6812X Compiler Compiler on PC
  - ZAP S12X SIM on PC
  - ZAP S12X Multilink BDM on PC
- Download the S12x demo kit
- Download 3rd party files for the S12X demo kit



Class Exercise #3 (con't)

- Unzip `s12x_demo_CM.zip` into a directory on your local hard drive.
  - This will be your `source` directory.
  - Read/write permissions for the directory are required.
  - There should be no spaces in the directory path name.
- Install Cosmic Software's "Freescale HCS12X and XGATE evaluation tools."
  - This will be your `cosmic` directory.
  - Install the "P/E Micro drivers," which are drivers for the hardware.
  - If you are installing the full version of the Cosmic S12X tools, install:
    - S12X and XGATE Compilers: S12X compiler
    - ZAP BDM S12X: S12X debugger
- Unzip `MW_s12x_demo_CM.zip` into a temporary directory and place the following files into these folders:
  - **source\cosmic**
    - `crtsx.s, link_def.lkf, vector_s12x.c`
  - **cosmic**
    - `clib.exe` (not needed for the full version of Cosmic Cross Development Tools)
- Add in the two system variables as shown on the right:
  - **CXLIB**: Point to the `cosmic\lib` directory
  - **CXTEMP**: Point to your temp directory



Creating an Embedded Target

- Create baseline
- **Create blockset**

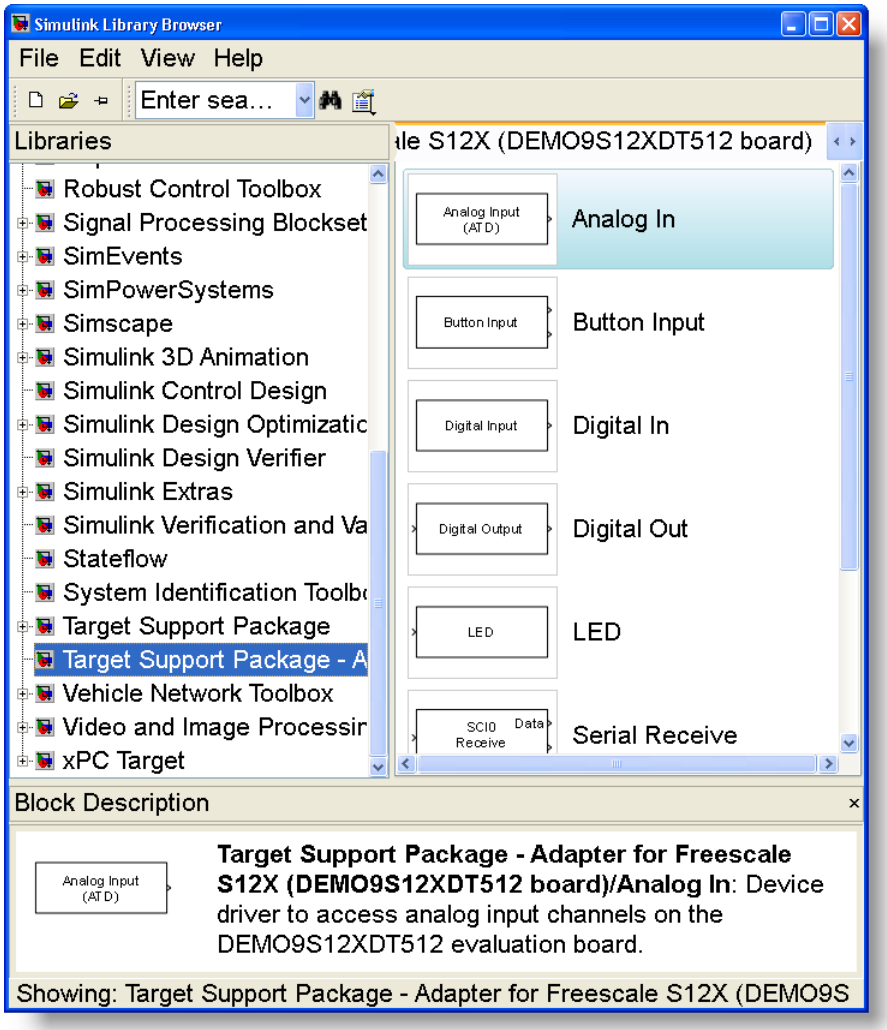
Creating Analog and Digital I/O blocks

General Device Driver Block Process

- Write a C function to define the driver data/model interface, such as the dimensions and data types of its inputs/outputs and parameters.
- Write an ASCII file to insert the read/write operations of the I/O device into the model algorithm during code generation.
- Define the necessary device driver block user interface.
- Place the device driver block block in a driver library of the target.

S12X Device Driver Blockset

- The device drivers are standard S-functions.
- S-functions are standard callable functions or inlined for efficiency.
- In the case of most S12x drivers inlining and direct access of the registers from the main program is used.
- For some drivers such as drivers that use interrupts, a C interrupt handler is used.
- The C interrupt handler communicates with the model code with global variable as shown above.

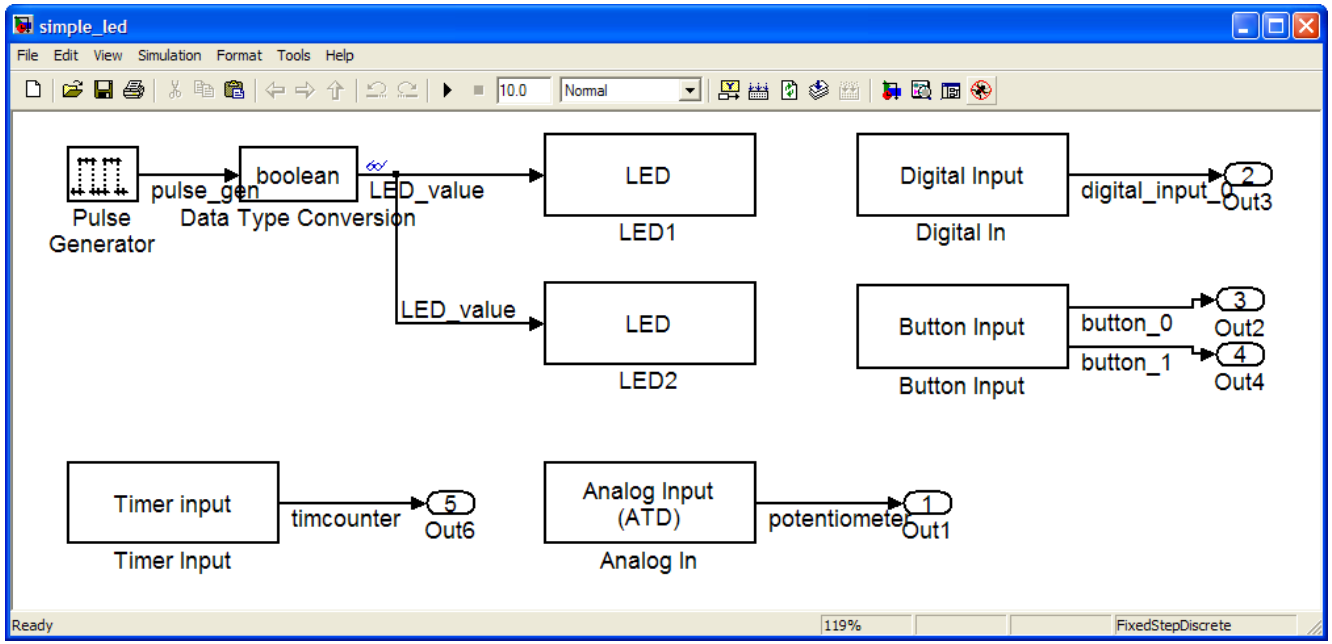


Class Exercise #4

- Examine the Digital I/O block
- Write an outline for how to develop LED driver block
- Observe the instructor's LED driver and example (next slide)

Live Example of LED Block

- Instructor led demo
- Observe hardware



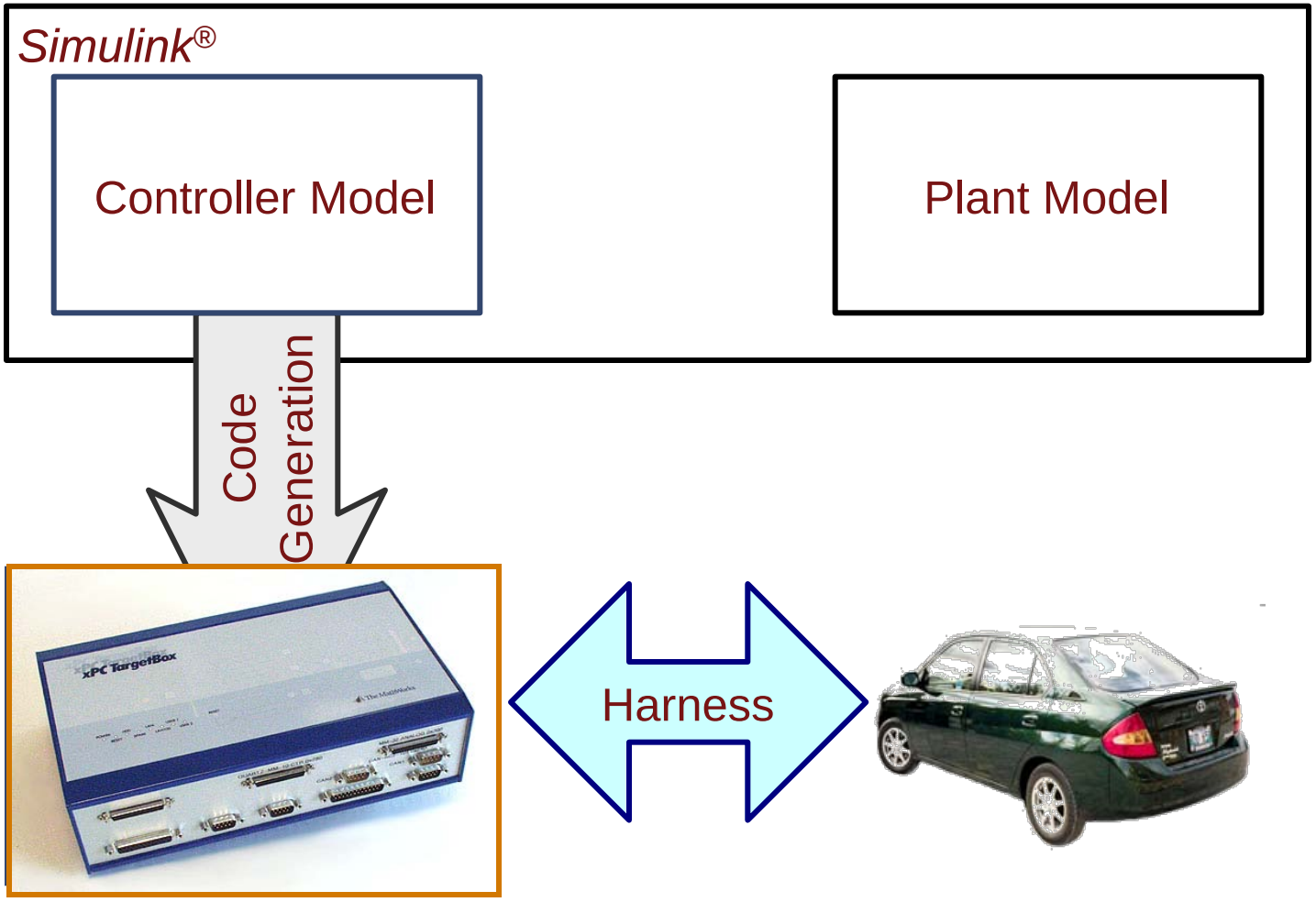
Agenda

I. Introduction (60 minutes)
II. Creating a Host Target (60 minutes)
III. Lunch
IV. Creating an Embedded Target (60 minutes)
V. Verifying an Embedded Target using PIL test (30 minutes)

Verifying an Embedded Target using PIL

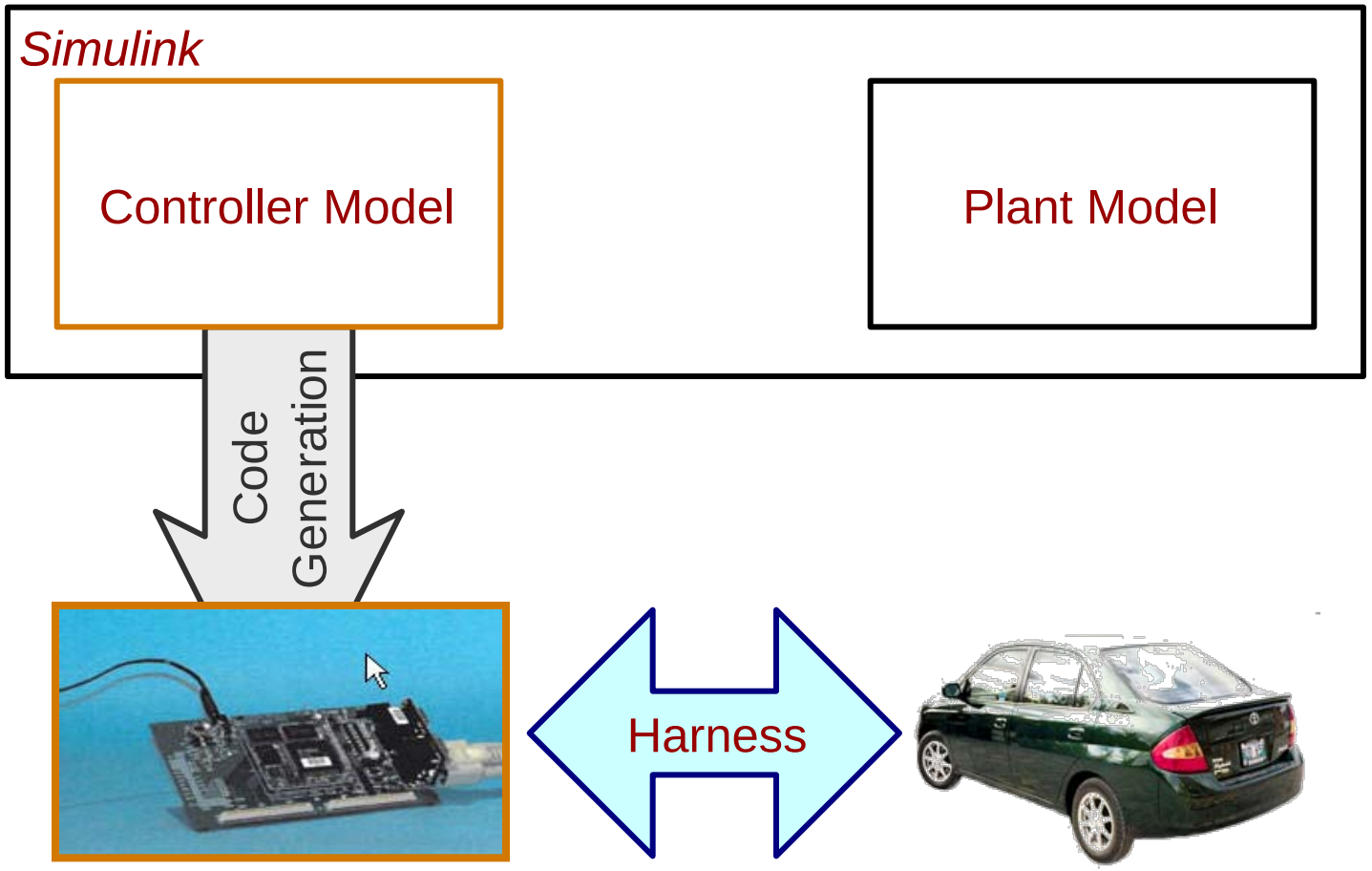
- **Introduction**
- Create PIL Application

Traditional Rapid Prototyping



xPC Target®

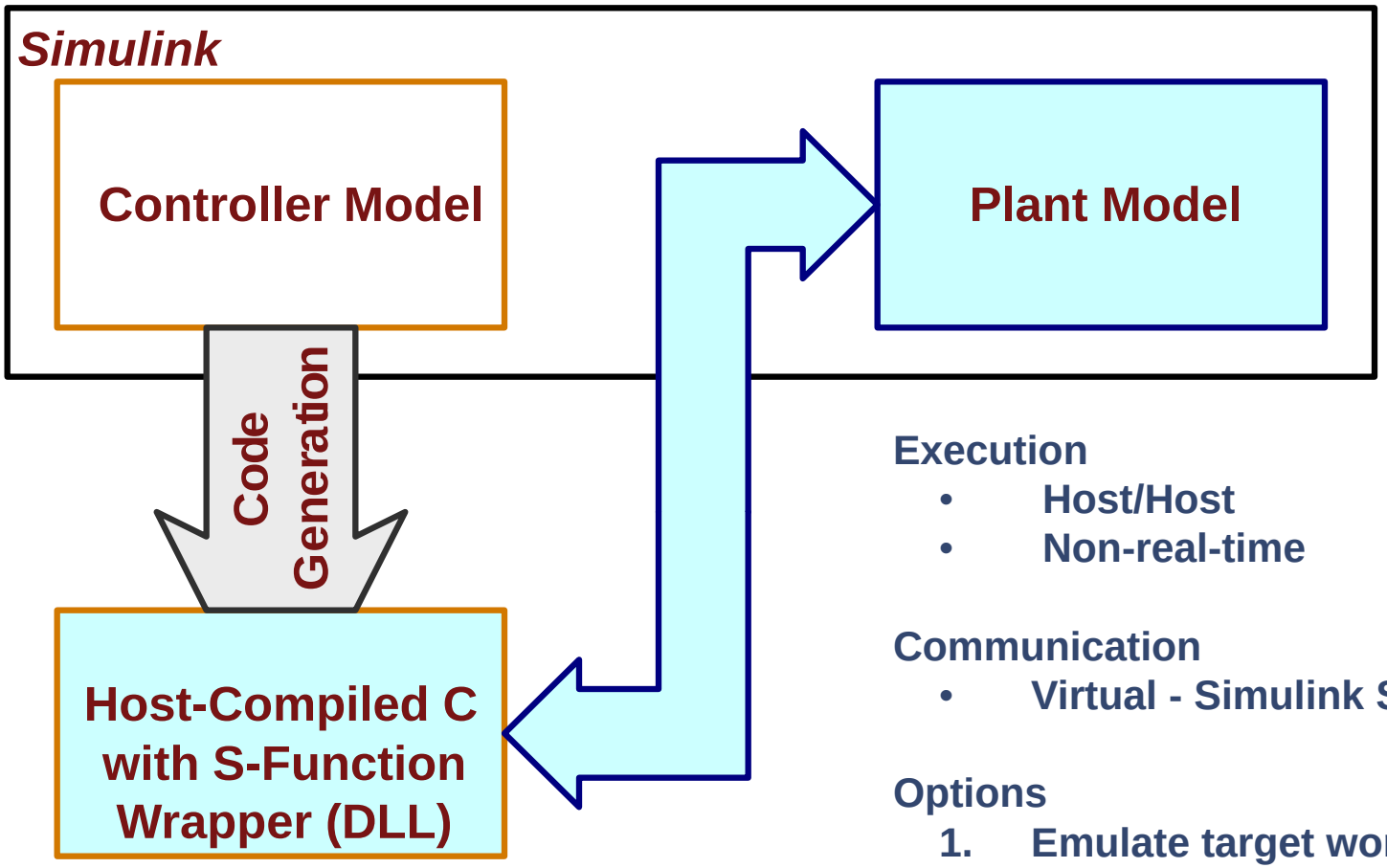
On-Target Rapid Prototyping



Rapid Prototyping Comparison

	Traditional	On-Target
Purpose	Useful for testing new ideas and green-field research	Useful for refinement and calibration of designs during development process
Execution Hardware	Uses PC or non-target HW	Uses ECU or near-production HW
Code Efficiency, I/O latency	Less emphasis on code efficiency and actual I/O latency	More emphasis on code efficiency and actual I/O latency
Programs	Works well for new vehicle programs	Works well for delta changes to existing programs
Engineers	Typically done by systems engineers in R&D or advanced production	Typically done by systems and software engineers in production
Cost and Convenience	May require custom real-time simulators and hardware, or may be done with inexpensive “off-the-shelf” PC hardware and I/O cards	May use existing hardware, thus less expensive and more convenient

Software-in-the-Loop Verification



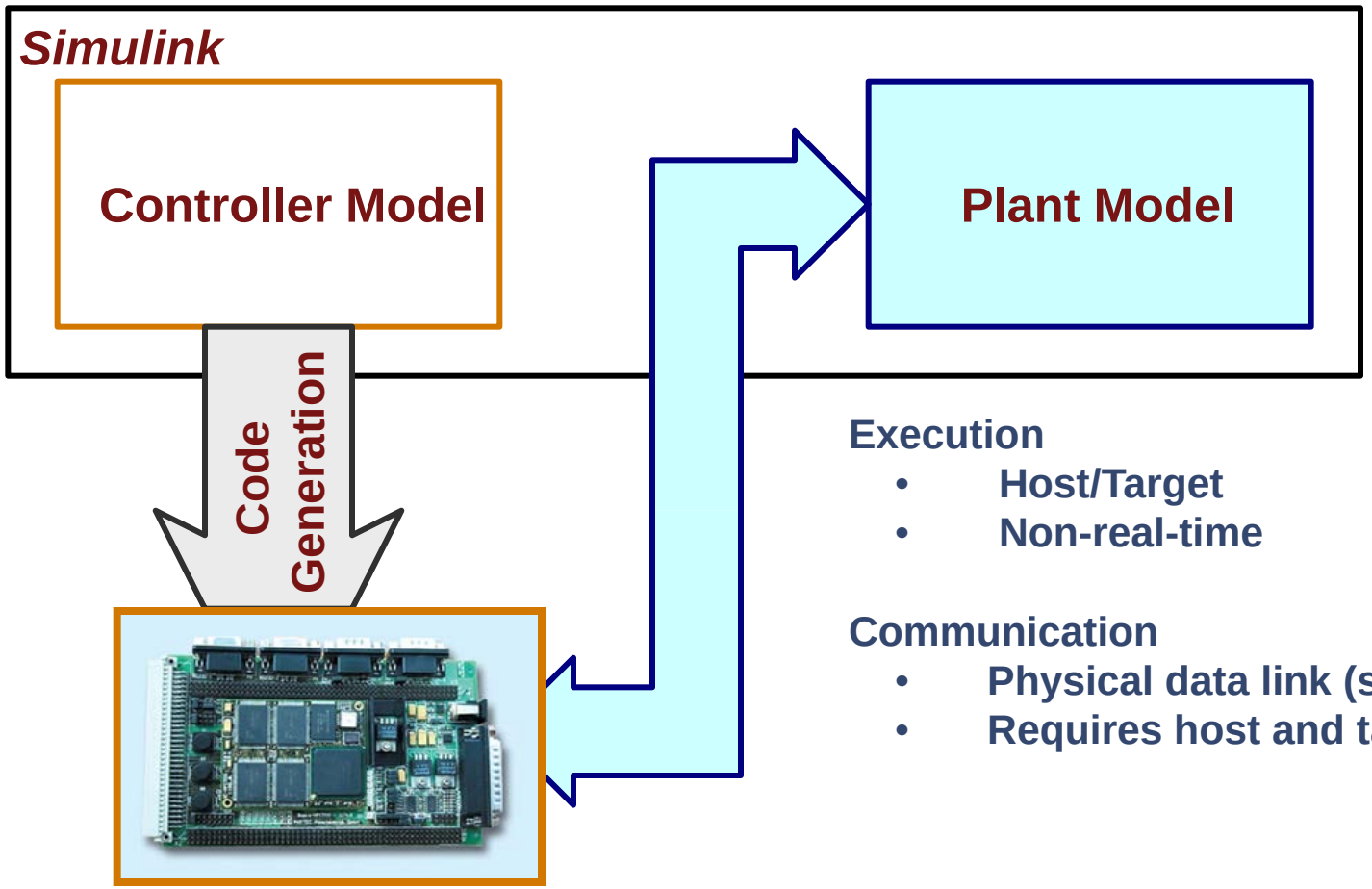
Execution
- Host/Host
- Non-real-time

Communication
- Virtual - Simulink S-function

Options
1. Emulate target word sizes (change code)
2. Use actual target word sizes (do not change code)

Processor-in-the-Loop Verification I

Direct Communication to Hardware



Execution

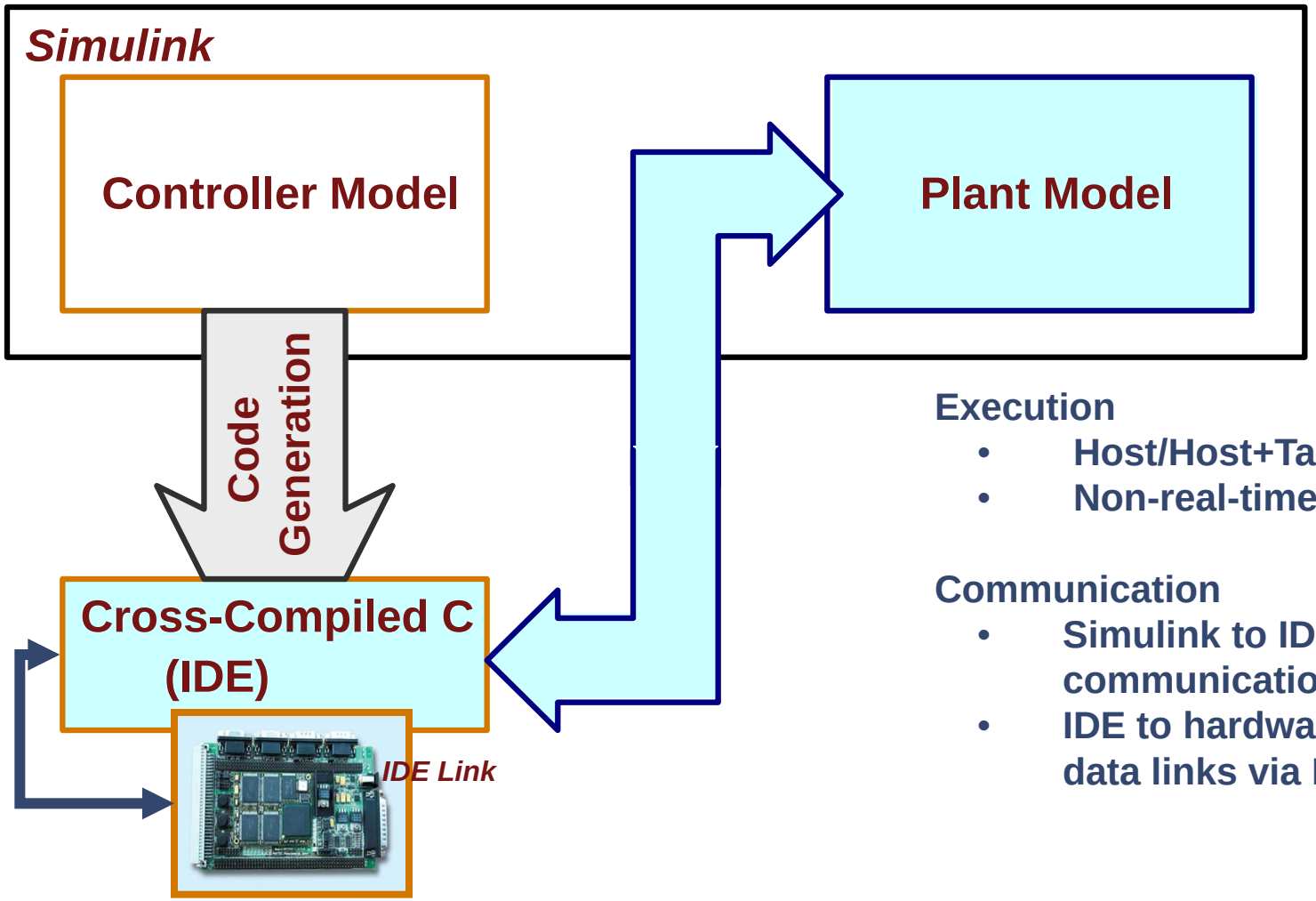
- Host/Target
- Non-real-time

Communication

- Physical data link (serial, CAN, …)
- Requires host and target drivers

Processor-in-the-Loop Verification II

IDE Communicates to Hardware



Execution

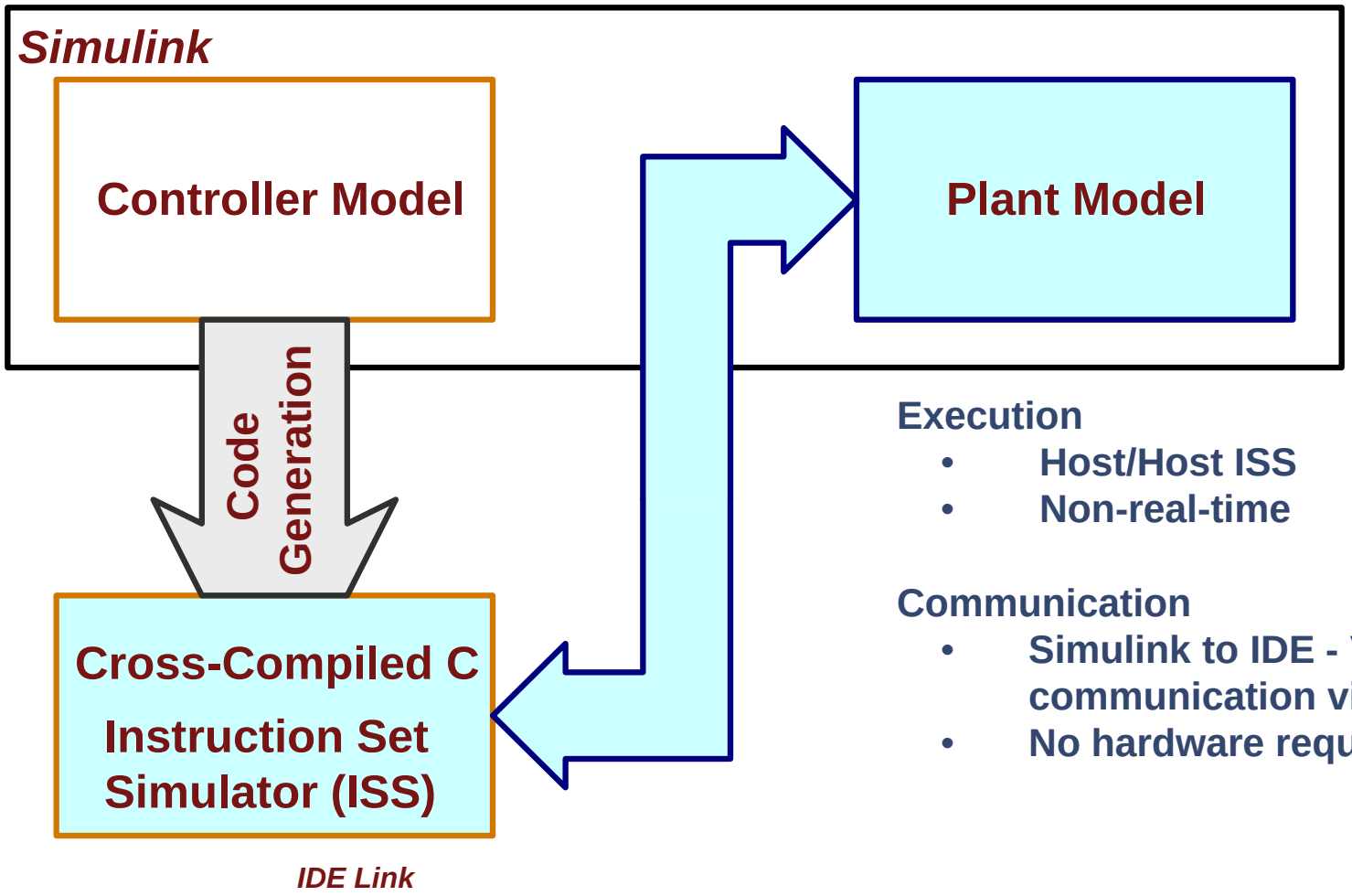
- Host/Host+Target
- Non-real-time

Communication

- Simulink to IDE - Virtual communication via PIL APIs
- IDE to hardware - Physical data links via IDE debugger

Processor-in-the-Loop Verification III

IDE Simulates Hardware



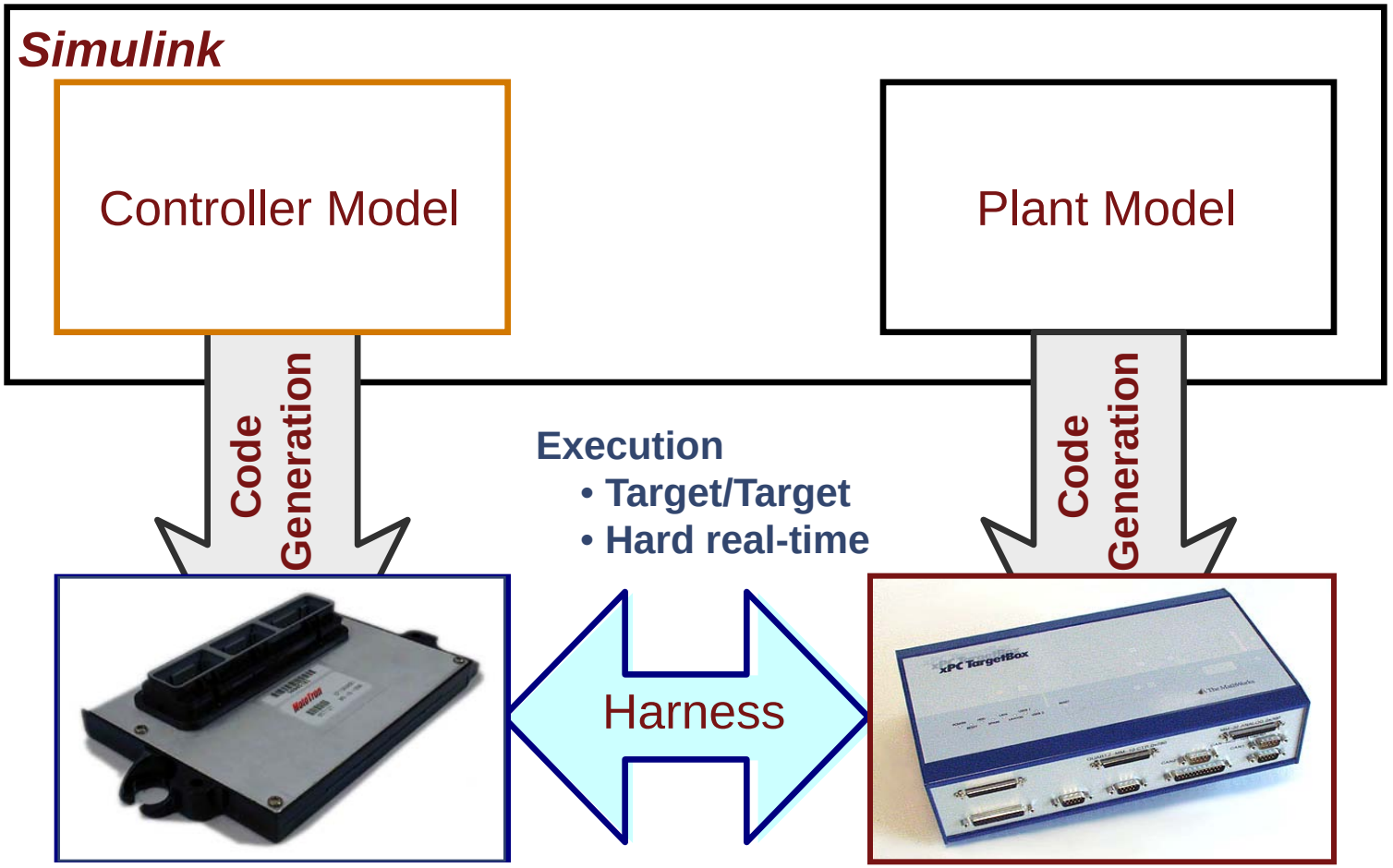
Execution

- Host/Host ISS
- Non-real-time

Communication

- Simulink to IDE - Virtual communication via PIL APIs
- No hardware required

Hardware-in-the-Loop Verification



Should you expect the same functional behavior in simulation and on the target?

For floating-point applications

- Slight numerical differences arise due to target-specific library functions and other effects.
- It is important to ensure that your algorithm is robust to implementation variations by running SIL, PIL, and using open and closed loop tests.

For fixed-point applications

- You should expect to achieve bit-level accuracy comparing simulation and on-target results.
- Simulink® Fixed Point™ lets you simulate bit-true behavior; convert float- to fixed-point math, and compare float to fixed results all within Simulink, without generating code.

Sometimes there may be expected or intentional differences, for example resulting from

- Use of target-specific libraries functions (for floating-point, trigonometric functions, etc.)
- Use of hardware-specific optimized code (e.g., use of a 40-bit accumulator)

Sometimes there may be unexpected differences, for example resulting from

- Unintended side effects of compiler settings and optimizations
- Unintended side effects of code generator settings and optimizations
- Defects in the hardware, the compiler, the linker, or the code generator
  - See MathWorks Bug Reports www.mathworks.com/support/bugreports/

Verification Approaches: Benefits

Benefits of PIL

- Target verification of individual software components (unit testing)
- Exercises the same *object code* that will be used in production software
- Verify correct behavior on real or simulated target hardware
- Evaluate on-target behavior with closed-loop plant model in Simulink
- Collect metrics for:
  - stack profiling
  - code coverage
  - execution profiling

Benefits of SIL

- Host verification of individual software components (unit testing)
- Exercises the same *source code* that will be used in production software
- Offers a subset of the benefits and metrics provided by PIL
- Easier to do than PIL

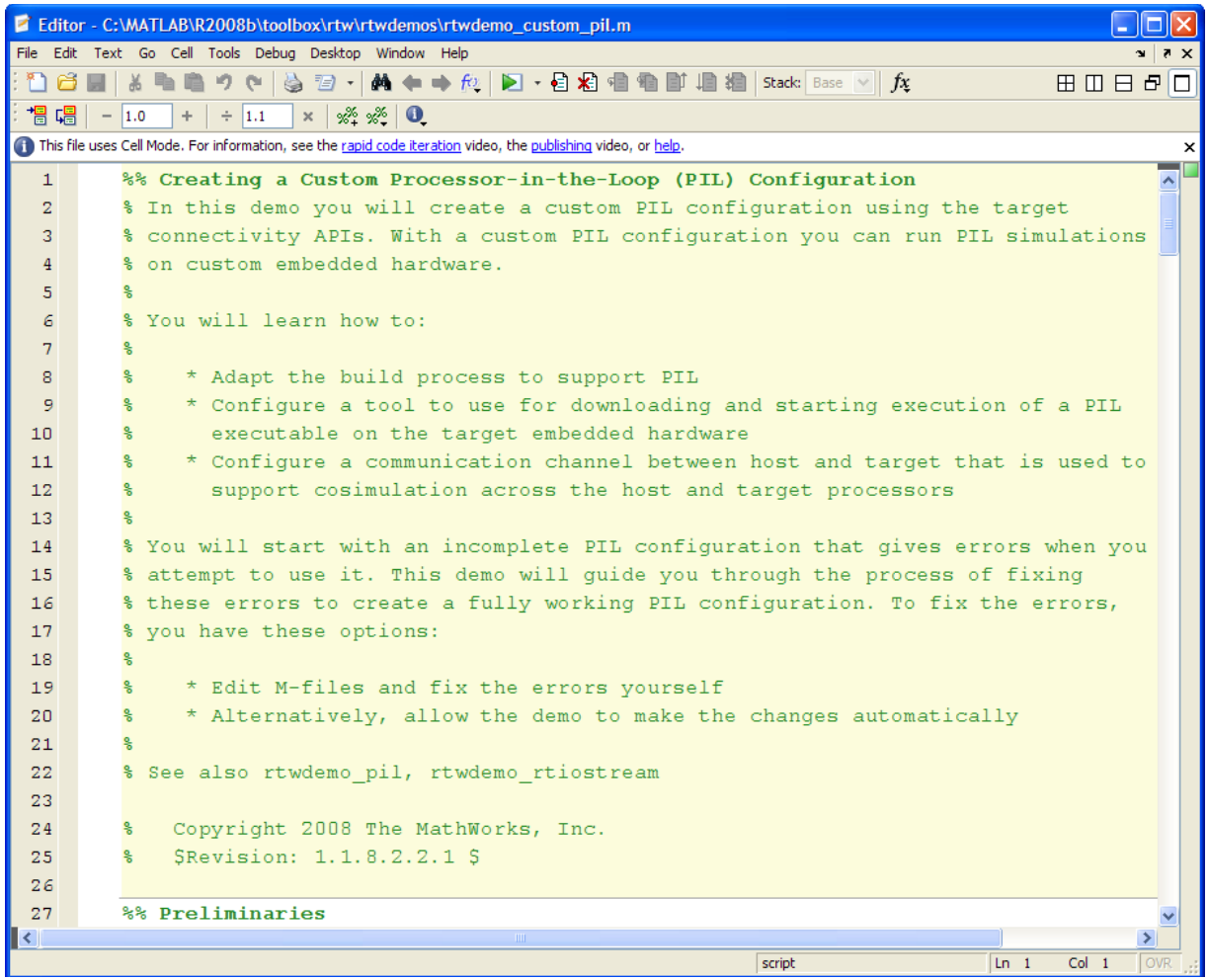
In-the-Loop Verification Comparison

	SIL	PIL I, II (HW)	PIL III (just IDE)	HIL
Purpose	Verify source code component	Verify object code component	Verify object code component	Verify system functionality
Fidelity and Accuracy	Option 1: Same source code as target but may have numerical differences Option 2: Changes source code to emulate word-sizes but is bit-accurate for fixed point	Same object code; bit-accurate for fixed point; cycle-accurate since it runs on HW	Same object code; bit-accurate for fixed point; may not be cycle-accurate since it runs in ISS not HW	Same executable code; bit-accurate for fixed point; cycle-accurate; use real and emulated system I/O
Execution Platforms	Host/Host	Host/Host/Target	Host/Host	Target/Target
Ease of use and cost	Desktop convenient; executes just in Simulink; no HW cost	Executes on desktop or test bench; requires HW for processor board and cables	Desktop convenient; executes just on host computer with Simulink and IDE; no HW cost	Executes in test bench or lab; uses HW for processor, ECU, I/O, cables
Real-Time	Non real-time	Non real-time (between samples)	Non real-time (between samples)	Hard real-time
Engineers	Systems or software engineers	Software or test engineers	Software or test engineers	System or test engineers

Verifying an Embedded Target using PIL

- Introduction
- **Create PIL Application**

Creating PIL Application

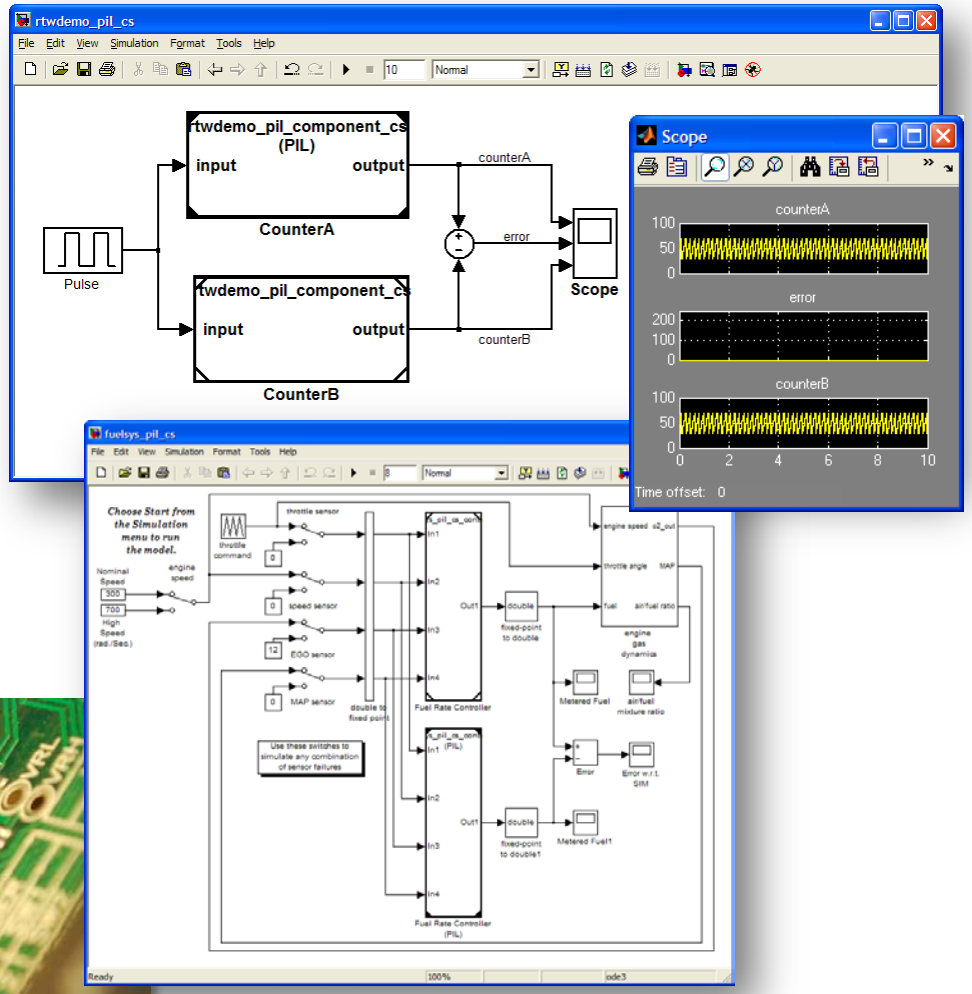


>>edit *rtwdemo_custom_pil*

Example PIL

PIL Demo (time permitting)

- `rtwdemo_pil_cs.mdl`
  - Simple model comparing simulation and PIL testing results
- `fuelsys_pil_cs.mdl`
  - Fuel Rate controller model comparing simulation and PIL results
  - Requires full license for Cosmic Cross Development Tools



Conclusions

- Embedded target integration is a key component of ECU development process
- There are many options from algorithm export to full featured embedded target
- We hope you learned some useful technologies
- Please contact MathWorks for obtaining latest S12x example target

  mytarget@mathwork.com

Next Steps

- Try *rtwdemos*
- Read “Developing Embedded Targets”
- Attend Training Classes and Workshops
- Contact your local MathWorks representative