Development of Real Time Systems using Simulink/RTW and RTLinux

MSEE Plan B Project
Steve Goetz
May 2005

Professor Voyles, Advisor

1 Project Overview

The purpose of this project is to provide an integration of the code generation facilities of Matlab/Simulink with an available real time execution environment (RTLinux) for the purpose of creating control systems for robots. This integration was then demonstrated by the development of a driver, Simulink library, and test models for a specific interface board (ServoToGo Model 2).

This work consists of 4 main components, each of which is described in following subsections.

1.1 Matlab - Simulink and Real Time Workshop

Matlab is a mathematical modeling and numerical analysis package that is commercially available (www.mathworks.com). This tool provides a block diagram-oriented modeling tool called Simulink as a key feature. Simulink provides many standard blocks (Figure 1) allowing basic mathematical operations and signal routing as well as advanced blocks suitable for control systems and other complex operations.

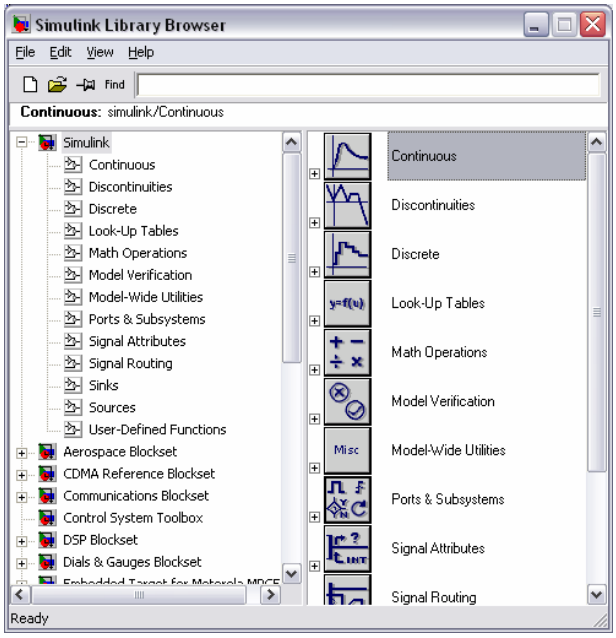


Figure 1 – Example of Simulink standard blocks

Models created in Simulink by dragging and dropping appropriate blocks are very useful for simulating complex systems. An added

feature of Simulink, called Real Time Workshop, extends this functionality by allowing the generation of executable code from a block model. This code can then be run on a target and used to control a physical plant (robot or other system).

1.2 Real Time Linux (RTLinux)

A second key component of this project is RTLinux. RTLinux is a set of patches that, when applied to a standard Linux distribution, allow the operating system to be real time deterministic without sacrificing the base functionality of the Linux world. This product is available from FSMLabs (www.fsmlabs.com) in both free (RTLinuxFree) and commercial versions (RTLinuxPro). This work uses RTLinuxFree with RedHat Linux 9.0 distribution and kernel version 2.4.20.

1.3 Integration Components

1.3.1 Simulink - RTLinux

Simulink's Real Time Workshop generates code from a model for a specific target environment. These can range from bare-board level (determinism provided via a timer interrupt service) to full RTOS multitasking support (determinism provided using preemption and semaphores). Each target is defined by a configuration file (*.tlc). This work leverages a previously developed integration package by Raul Murillo Garcia for lab PC process control and measurement at Glasgow Caledonian University (http://www.sesd.gcal.ac.uk/raulm/St-rtl.htm#Introduction) that provides the basic execution engine for the RTW generated code.

1.3.2 ServoToGo - RTLinux

Currently, access to the ServoToGo interface board from the RTLinux environment is provided via a character device driver developed specifically for the purpose. This driver provides a very low level interface to the hardware – taking in address read/write level commands, performing the actual memory reads/writes, and then returning the results. A set of utility functions, included as part of the Simulink generated code, exist to facilitate this interface.

While this architecture has some advantages in ease of development and portability across platforms, it also has some significant performance implications. Alternate implementation options include a driverless architecture, wherein the Simulink generated code directly

accesses hardware, or a more capable driver model, wherein the driver manages a higher level of functionality, are both possible in order to improve performance.

2 How to Use

This section describes in detail how to build a model, generate code from it, compile the code on the target, and execute the model.

2.1 ServoToGo Library

The ServoToGo library developed during this project contains blocks for analog input and output, encoder reads and writes, and initialization of the interface board (Figure 2).

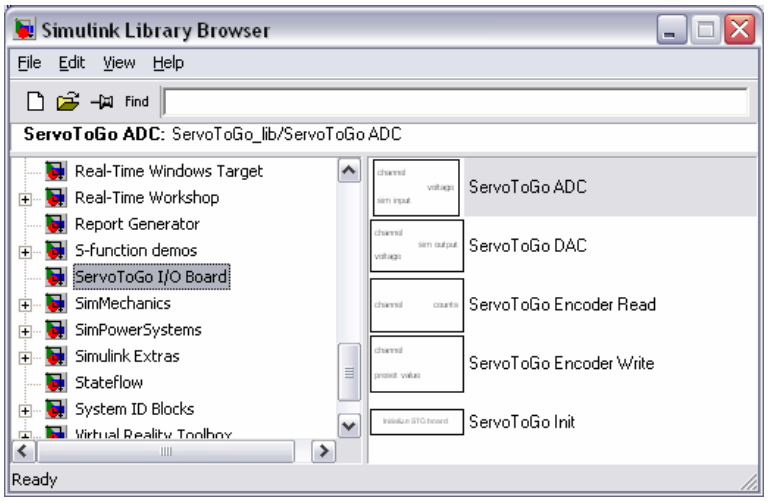


Figure 2 – ServoToGo library blocks

Note that, in order for the library to be available, the path of the library components must be added to the Matlab path prior to opening the Simulink library browser. This can be accomplished using the following command from the Matlab command line:

path (path, '<matlabroot>\rtw\c\servotogo')

where <matlabroot> is the root directory for Matlab (c:/matlab6p5)

2.2 Creating a Model

First, one would create a model. An example model is shown in Figure 3.

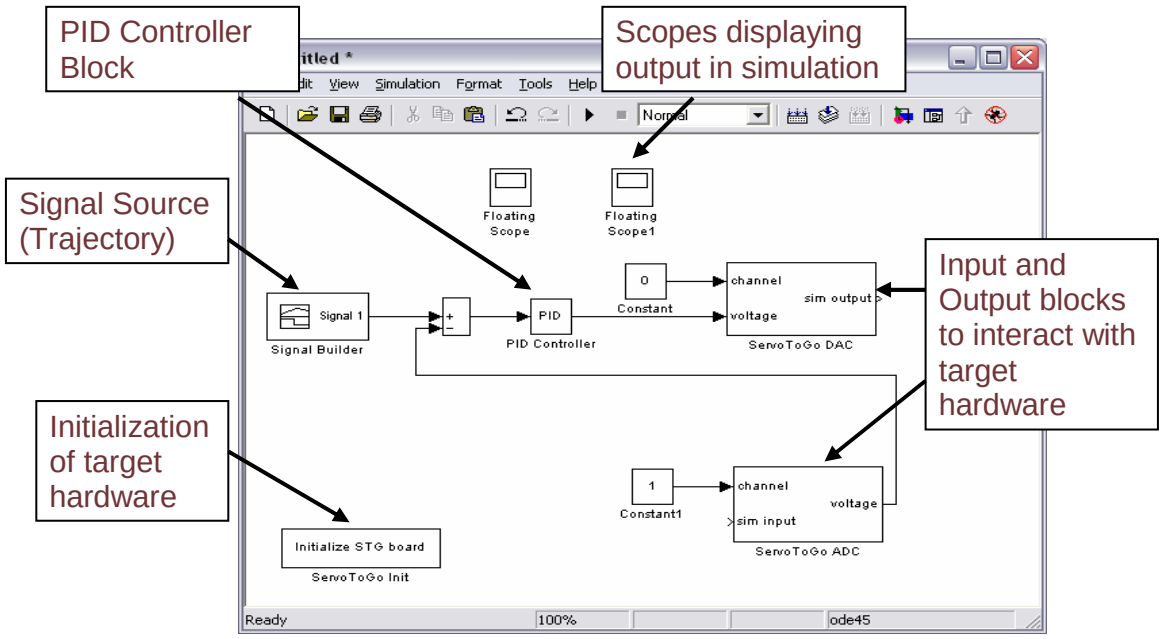


Figure 3 – Example model containing a PID controller

2.3 Configuring the Model

Prior to code generation, the model must be appropriately configured. First, the correct target config file (*.tlc) must be selected (Figure 4).

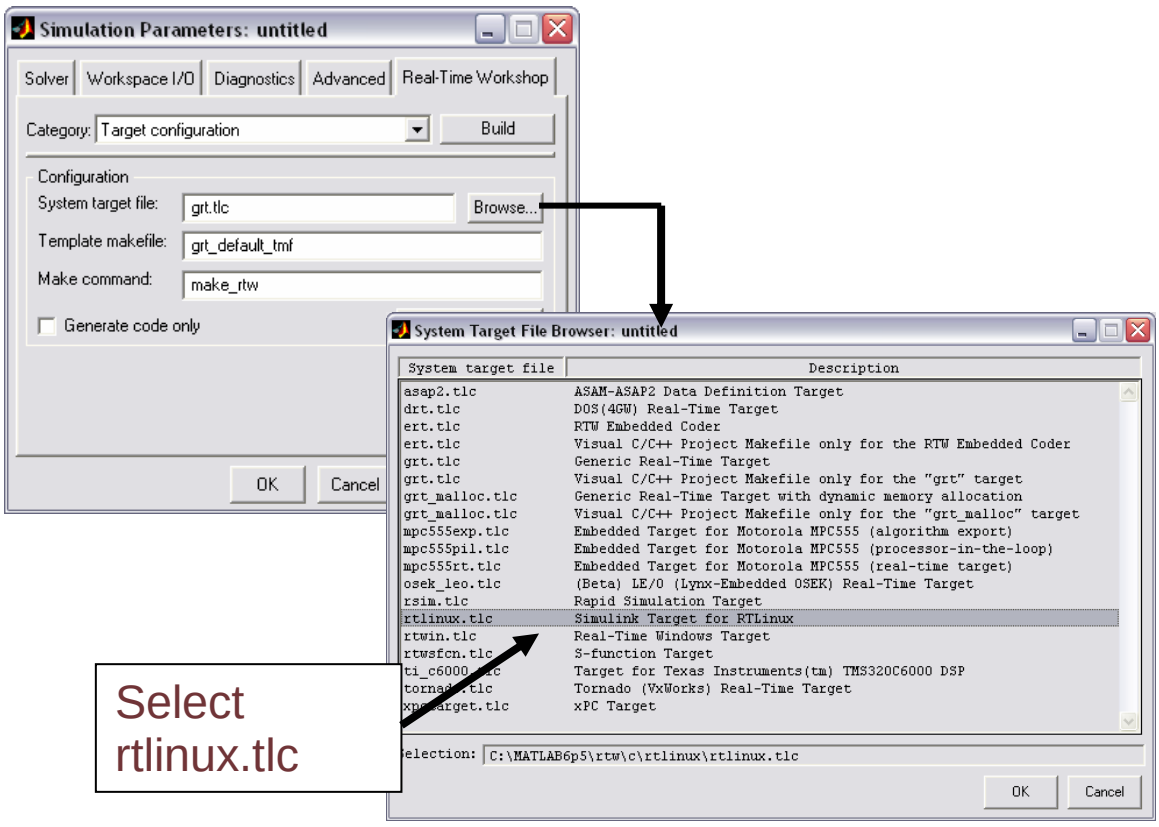


Figure 4 – Selection of rtlinux target

Additionally, the model parameters controlling the step rate and solver options must also be selected (Figure 5).

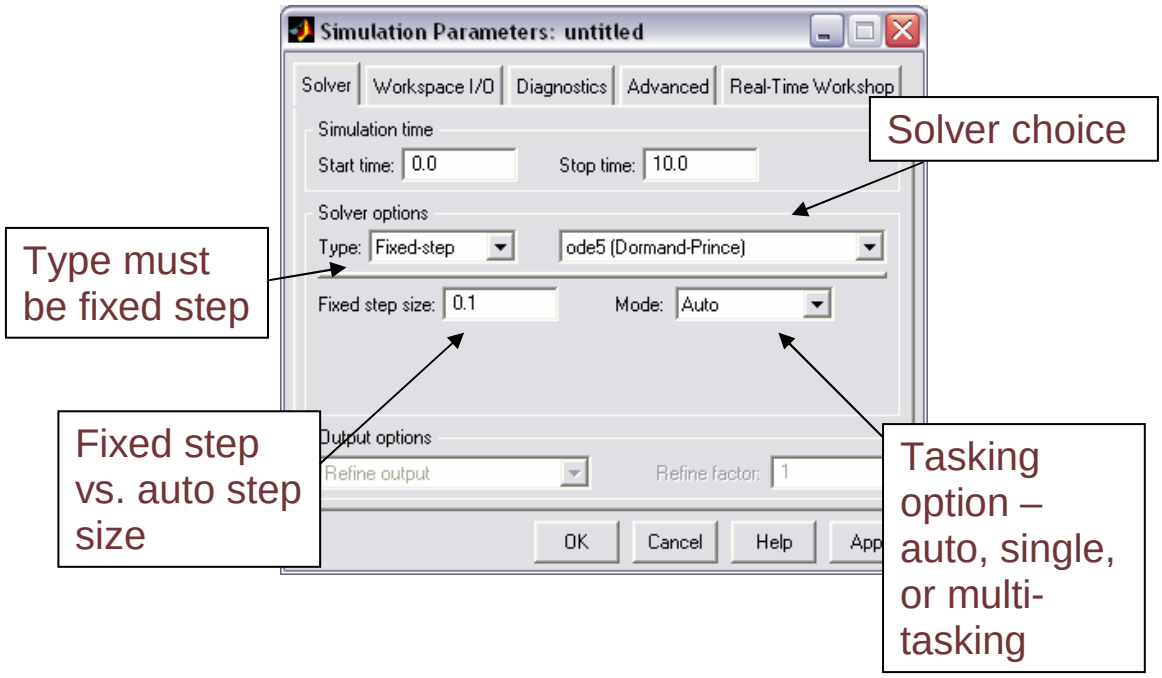


Figure 5 – Model execution options

For RTW code generation to work, the model must use a fixed-step solver.

A fixed step size is likely preferable (for ease of debugging and determinism) to an automatically generated step size. Also, this can be used to control the base rate of the model.

Solver choice is dependent on the type of blocks used in the model. If only discrete time step blocks are used, the discrete solver can be used. For models using continuous blocks, one of the other solvers must be used.

Finally, RTW allows for the specification of a tasking model. Single tasking runs the model as a single task. This places some limitations on the base rate of the model (base rate must accommodate worst case step – all blocks that must execute must execute within the base period). Multitasking allows additional flexibility, and integrates well with RTOS execution. The RTLinux system supports multitasking models.

2.4 Generating Code

With the target and model configured, code can be generated (Figure 6). For execution on the target, it is generally sufficient to generate code only.

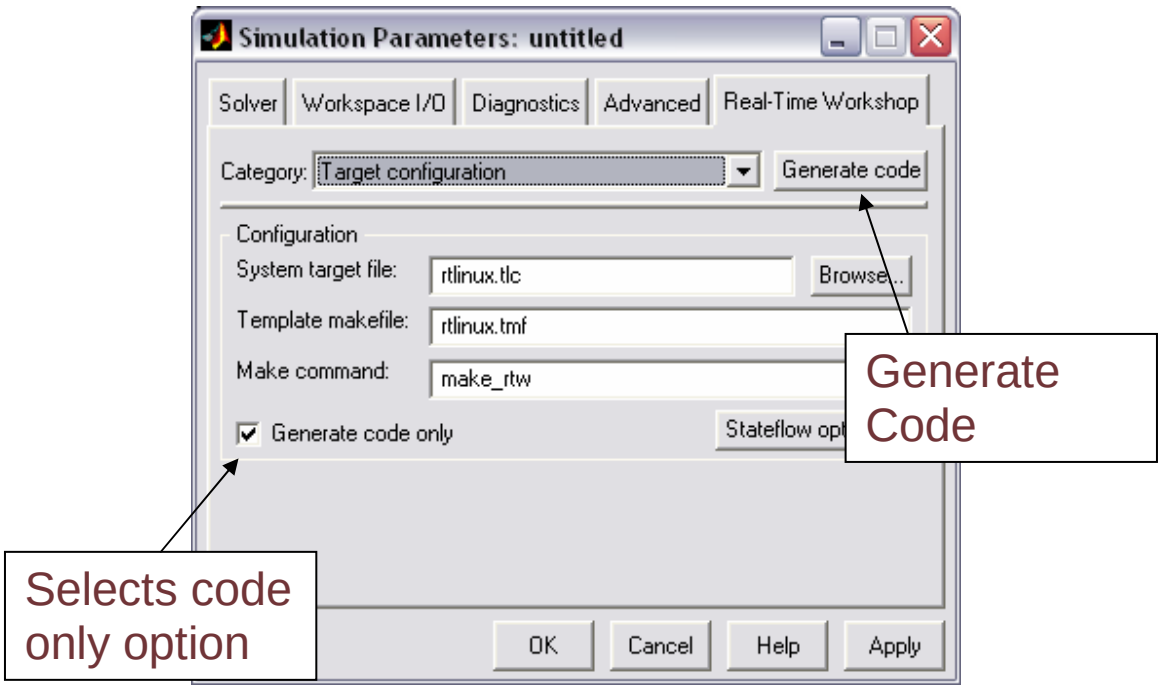


Figure 6 – Code Generation.

2.5 Results of Code Generation

Code generation results in a set of files describing the model. The base name for the files is the name of the project – in the example, 'demopid'.

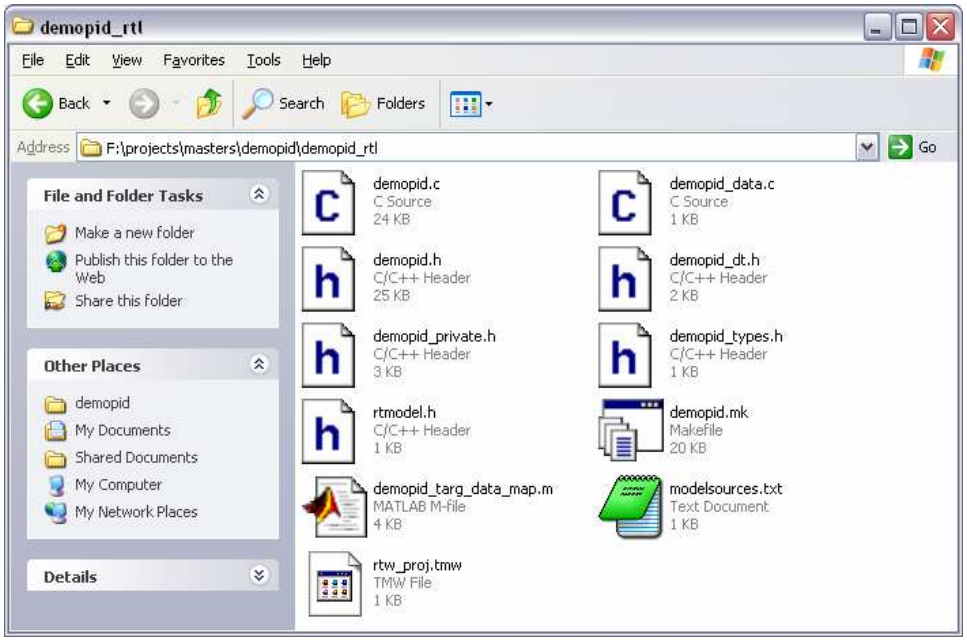


Figure 7 – Files generated by Real Time Workshop

The key files for execution on the target include only the sources (*.c, *.h) and the make file (.mk).

2.6 Target Compilation

The sources and the make file can now be transferred to the target (typically by zipping them and transferring via FTP, although any other means will work).

Once on the target, unzip the files into a directory. From that directory, make the executable.

make –f projectname.mk

This will make the generated code and, if successful, start RTLinux and execute it. Prior to doing this, the driver should be installed and RTLinux should be started.

3 How to Install

This section describes installation and setup for a system consisting of a Win32 development machine (Matlab/Simulink) and a Linux target.

3.1 RTLinux Install and Build Process

This work used the RedHat Linux distribution version 9.0. Begin by installing an appropriate configuration of RH Linux. Most options don't matter, but do make sure to include the development tools and libraries for 'c', the kernel development tools (which include the 'tcl' tools for xconfig), and the kernel source.

Note – much of this must be done from the root account!

Patch Kernel and Build Image

1. Download kernel sources: ftp.kernel.org -> /pub/linux/kernel/v2.4/linux-2.4.20.tar.gz
2. Make a directory -> mkdir /usr/src/rtlinux
3. Unpack the sources into /usr/src/rtlinux: *tar –xfz linux-2.4.20.tar.gz*
4. Change to the new linux directory: *cd linux-2.4.20*
5. Unpack the RTLinux archive into /usr/src/rtlinux/*rtlinux-3.1/kernel_patch-2.4.20-rtl3.2pre2: tar –xf rtlinux-3[1].2-pre2.tar*
6. Apply FSM Lab's RTLinux patch: *patch –p1 < /usr/src/rtlinux/rtlinux-3.2-pre2/patches/kernel_patch-2.4.20-rtl3.2pre2*
7. In /usr/src there is a link to the kernel directory (linux-2.4). Change this link to point at the new kernel directory *(/usr/src/rtlinux/linux-2.4.20*)
8. It is easiest to start with one of the RedHat .config files. Copy the most appropriate (for example, *kernel-2.4.20-i686.config*) from */usr/src/linux-2.4.20-6/configs* to the new linux directory (*/usr/src/rtlinux/linux-2.4.20*) and change its name to '*.config'*
9. Make a new config: *make xconfig*
10. In the new config, disable APM (under '*General setup'*). This can interfere with RTLinux.
11. From the new linux directory, */usr/src/rtlinux/linux-2.4.20*, make dependencies: *make dep*
12. Make the kernel: *make bzImage*
13. Build any remaining modules: *make modules*, *make modules_install*
14. Make the boot ram disk image: *mkinitrd /boot/initrd_rtlinux.img 2.4.20-rtl3.2-pre2*

15. Move the kernel and associated files to the proper locations: *make install*
16. Open grub.conf (at */boot/grub/grub.conf*) to verify that the new kernel is available at boot time. There should be a new entry for a kernel called *Red Hat Linux (2.4.20-rtl3.2-pre2)*. Rename this (if desired) to reflect RTLinux appropriately.
17. Reboot and select RTLinux kernel from the grub startup menu

Make RTLinux Modules

1. From rtlinux directory: *cd /usr/src/rtlinux/rtlinux-3.2-pre2*
2. Create a symbolic link: *ln – sf /usr/src/rtlinux/linux-2.4.20 linux*
3. Remake the config: *make xconfig* (first accept the license agreement, then save and exit)
4. make dependencies: *make dep*
5. make RTLinux: *make*
6. Run the script: *sh /scripts/insrtl*
7. *make devices*
8. *make install*

3.1.1 Start RTLinux

1. From terminal: *rtlinux start*
2. The system should tell you that the RTLinux modules were successfully started.

3.2 RTLinux Integration Install

The target integration package must be installed on the RTLinux machine and on the development machine.

The package is distributed as a zipped archive STRTL_M65_v1.4.zip, available from http://www.sesd.gcal.ac.uk/raulm/St-rtl.htm#STRTL.

Full instructions are included in a readme in the distribution. A summary set of instructions follows:

On the development machine:

1. Unzip distribution in matlab directory structure: \(matlab root)\rtw\c\rtlinux
2. Add this to matlab search path ('path' command)
3. Restart matlab. Rtlinux target (rtlinux.tlc) now shows up in target selection list.

On the RTLinux machine:

1. Uncompress the archive tar -zxvf STRTL_M65.tar.gz to get STRTL_M65 directory
2. Copy this to match location of the macro MATLAB_ROOT in the template makefile rtlinux.tmf on the development machine (typically */STRTL_M6.5*).
3. Fix two build issues:
    a. In */STRTL_M6.5/rtw/c/src/common.h*, fix the #endif on line 102 by commenting '_COMMON_H' with '//'
    b. In */STRTL_M6.5/rtw/c/src/knrl_main.c*, remove the first '*' (immediately following the '*(hrtime_t)*' cast)

In order to include the servotogo library in the default makefile for the generated code, the default template make file for the RTLinux integration must be updated.

On the development machine:

1. Delete the template make file rtlinux.tmf (located in (matlab root)\rtw\c\rtlinux)
2. Copy the STG variant provided (rtlinux stg.tmf) into this directory and rename it as rtlinux.tmf

3.3 ServoToGo Library

Code for this library (and other custom libraries) must be located on both the target and the development machine. On the host, the code and associated library files are necessary to support the availability of the blocks in the Simulink library and to allow for simulation of the model. On the target, the code is compiled as part of the model compiling process.

Development Machine

1. Unzip the stg_lib_rtl_host.zip archive from the Matlab root directory (C:\Matlab6p5)
2. Verify that files end up in <matlab root>\rtw\c\servotogo
3. Start Matlab and add this install directory to the path with the following command:
   path (path, 'c:\matlab6p5\rtw\c\servotogo')
4. Start Simulink. ServoToGo library should now be available

Target Machine

1. Unzip stg_lib_rtl_target.zip into /STRTL_M6.5
2. Verify that the files end up in /STRTL_M6.5/rtw/c/servotogo

3.4 ServoToGo Driver Install

The driver builds and installs as a separate component.

Building the Driver

1. Unzip the driver code into a directory: *unzip stg_driver.zip*
2. Verify that two files are created – stg_driver.c, stg_driver.mk
3. Make the driver: *make –f stg_driver.mk*
4. Verify that this results in an object – stg_driver.o

The driver owns the address of the ServoToGo board, and must be compiled to match the hardware. This can be done by changing a #define in the stg_driver.c file. Also, a #define exists to enable/disable logging of reads and writes. Logging is helpful for debugging, but can be cumbersome for actual execution.

Installing the Driver

1. Start RTLinux: *rtlinux start*
2. Insert the driver's module: *insmod stg_driver.o* (note that '*insmod*') is only available as a root user)
3. Verify that it inserted correctly: *dmesg* (log should show "Servotogo registered with ID" where ID is the assigned driver ID)

For general debugging, the driver outputs many of its commands/responses to the system log. *dmesg* prints the log to the console, and *dmesg –c* clears it (after printing it to the console).

4 How to Extend – Creating Blocks and Libraries

Simulink provides tools for creating new blocks and additional libraries.

4.1 Block Creation

Block creation facilities are fully documented in the Matlab help files. The following is intended only to be a brief overview and how-to.

Simulink provides a method for creating blocks based on Matlab s-functions. The block basically masks the s-function, passing along inputs to it and taking outputs and feeding them back into the model.

s-functions themselves are bits of functionality implemented in another programming language (typically c or c++), and then interfaced to the Simulink environment through support for a set of standard callback functions.

There are two versions of s-functions supported – non-inlined and inlined. These differ in how they are treated on the workstation (under simulation) and on the target. Inlined functions have separate implementations – a standard, callback based version used in simulation and a streamlined implementation appropriate for the target that is substituted into the target implementation in-line. Non-inlined versions execute the same standard model in both implementations. Non-inlined s-functions require more memory and increase execution overhead when compared to the in-lined version, but require only a single implementation to be developed and maintained.

To build a non-inlined block, one can begin with a provided template that contains all necessary callback functions. This is located in the standard Matlab install at C:\MATLAB6p5\simulink\src\sfuntmpl_basic.c (sfuntmpl_doc.c for a documented version).

Code appropriate to the block functionality can then be added to the callback functions – typically in the mdlStart(), mdlUpdate(), and mdlTerminate() functions, but possibly elsewhere.

Code that is only appropriate on the target can be conditionally included using the following construct:

#ifdef RT

//target code
#endif

With the s-function code complete, a block can be created in Simulink. In the standard library under 'User Defined Functions' there is a basic s-function block. Add this block to a model, and rename it with the name of the c-file implementation – the implementation and the block are linked via this shared name.

Finally, to make simulation possible, a compiled version of the s-function must exist. Change the working directory to that containing the c-file implementation and compile using *mex –g sfunctionfilename.c.* This creates a .dll version of the s-function.

To make an inlined block, the easiest method is to use the S-Function builder block provided by simulink under the 'User Defined Functions' option in the library. This provides a wizard like interface for defining a block, and was used to build the ServoToGo blocks for this project.

4.2 Library Creation

A library in Simulink is simply a model containing the blocks in the library along with an .m file defining the library attributes (*slblocks.m*). Add a group of blocks to a new (empty) model and save the model with an appropriate name. Copy a version of slblocks.m from an existing library to the new library, and edit it with the correct file names and other information. Add the directory containing the model, the .m file, and the implementations of the blocks (and their mex-compiled versions) to the matlab path and restart Simulink. The new library should show up in the browser.

5 Appendix

5.1 Contents of Archive

STRTL_M65.tzr.gz	Archive of Simulink/RTLinux integration
Rtlinux-3[1].2-pre2.tar	RTLinux distribution
stg_lib_rtl_target.zip	Servotogo simulink block code for target
stg_lib_rtl_host.zip	Servotogo library and code on host
rtlinux stg.tmf	Template make file to replace that found in distribution (STRTL_M65)
stg_driver.zip	Archive for STG driver
stg_lib_test_mdls.zip	Models used to test STG library

5.2 ServoToGo Driver Details

The interface to the ServoToGo board is implemented as an RTLinux character driver. It implements the following functions and file operations:

Functions:

```
init_module(): registers the driver as a character device
cleanup_module(): unregisters driver
```

File ops:

```
rtl_servotogo_open(): protects against multiple opens with a semaphore
rtl_servotogo_release(): releases open semaphore

rtl_servotogo_read(): reads either a byte or a word from an address
                      offset

rtl_servotogo_write(): writes either a byte or a word to an address
                       offset
```

For reads and writes, the address offset, type of operation (read vs. write), and data value (writes only) are stuffed as ascii text into the char buffer sent in the write/read file operation.

For all operations, the driver itself knows the base address of the board. This is currently compiled in, but could be configurable at driver installation.

The intention is that this driver contains all of the target specific hardware detail (address, etc.) and performs all actual hardware interface (writes and reads).

5.3 Simulink Block Implementation

Each s-function is comprised of a set of callback type functions that actually implement the block. For this project's blocks, only the following subset of callbacks are important:

```
mdlInitializeSizes(): sets up number and type of inputs and outputs.
                      For the initialize block, this also opens the
                      board driver and stores its handle.

mdlInitializeSampleTimes(): sets up number and type of sample times

mdlStart(): performs one time initialization

mdlOutputs(): performs actual work for the block (reads ADC, outputs
            DAC voltage, reads encoder, etc.)

mdlTerminate(): performs any clean-up (closes driver, etc.)
```

Each block's implementation resides in a pair of .c files as follows:

- *blockname*.c
- *blockname*_wrapper.c

The *blockname.*c file contains the callback functions mentioned above, and is used to generate the executable block for simulation on the host. The *blockname*_wrapper.c module contains wrapped versions of target specific code. These are called directly on the target machine.

In Simulink, these blocks are each wrapped in a mask (that provides input/output and block names) and are collected together in a library.

In addition to the blocks' code, there is a single helper file with various utility functions 'drv_utils.c'. This is present only on the target, and performs the actual interface to the hardware. This contains the following functions:

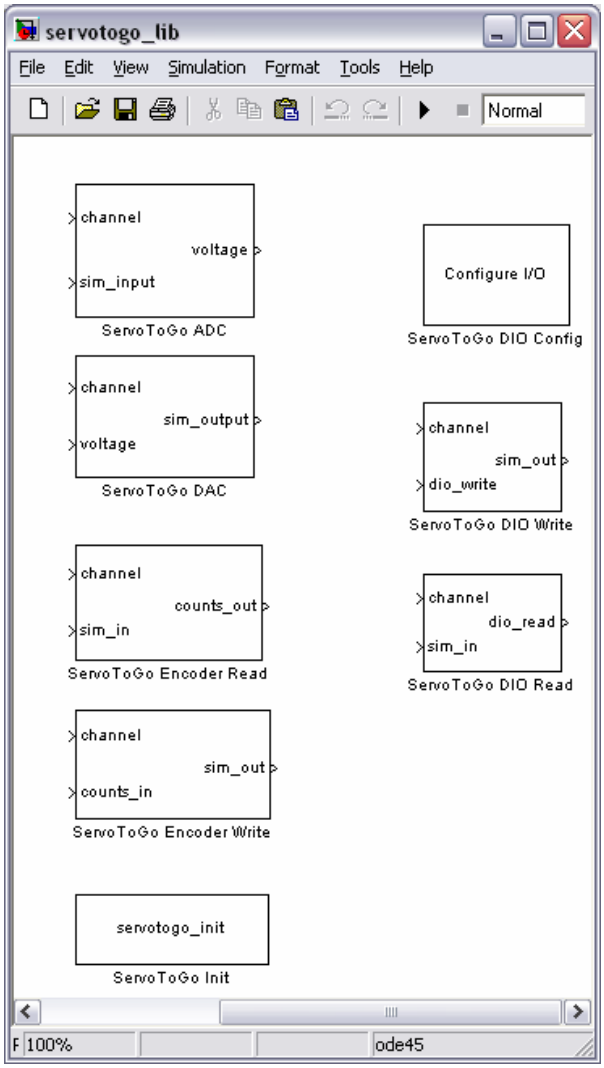
```
//i/o write/read wrapper prototypes
int drv_write_byte (int fd, unsigned char data, short address);
int drv_write_word (int fd, short data, short address);
unsigned char drv_read_byte (int fd, short address);
unsigned short drv_read_word (int fd, short address);

//board functions
```

```
int init_board (int fd);
int open_board (void);
void close_board (int fd);
void output_dac (int gFD, int channel, double voltage);
double input_adc (int fd, int channel);
double output_encoder (int fd, int chan, short *val);
double input_encoder (int fd, int chan);
void config_dio (int fd, int dirA, int dirB, int dirC0,
                 int dirC1, int dirD0, int dirD1);
void write_dio (int fd, int channel, unsigned char value);
unsigned char read_dio (int fd, int channel);
```

The drv_write/drv_read functions wrap the writes/reads to the actual I/O driver. The other functions wrap the details of their respective functions (ADC reads and scaling/conversion, etc.) and are the only things actually called in the s-function code.

5.4 STG library detail



Details of the ServoToGo library.

All blocks take a channel input.

The ADC returns a voltage read from the ADC channel as a double.

The DAC outputs a voltage input as a double to the specified DAC channel.

The Encoder read currently returns the counts it reads (not radians). The Encoder write block presets one of the encoder channels to a specified offset (in counts).

The Config DIO block has parameters that set up the DIO ports as either inputs or outputs. DIO read reads a value from a port

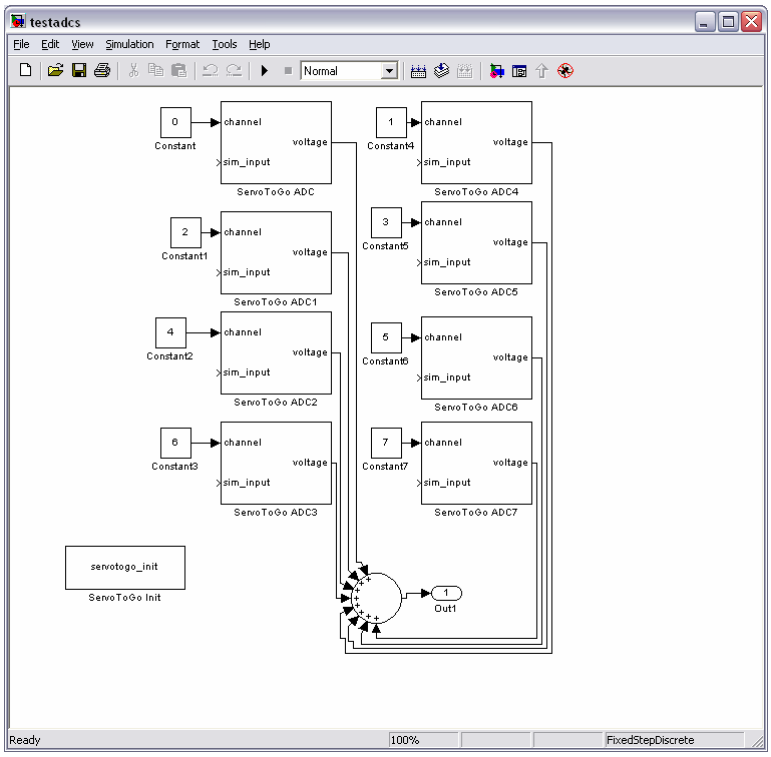
configured as an input, DIO write writes a value to a port set as an output.

The initialize block opens the board driver and performs initialization. This initialization currently consists of setting up the encoder registers, but could include digital I/O configuration as well.

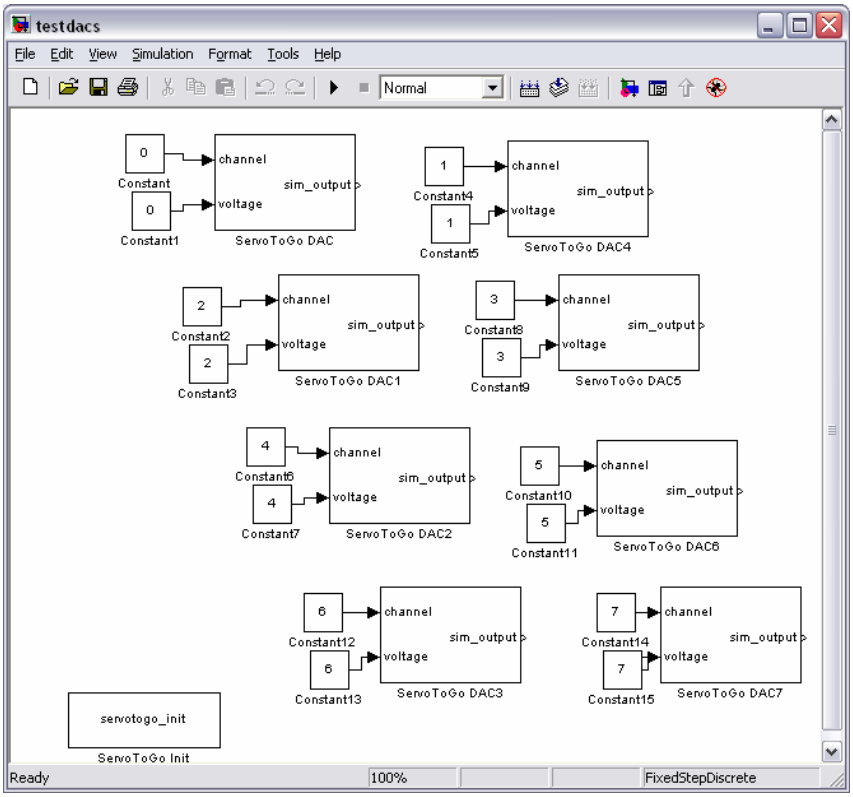
Some blocks also support simulation. When running in Simulink (not on the target), they use the ‘sim input’ and ‘sim output’ connections to pass data through.

5.5 Testing

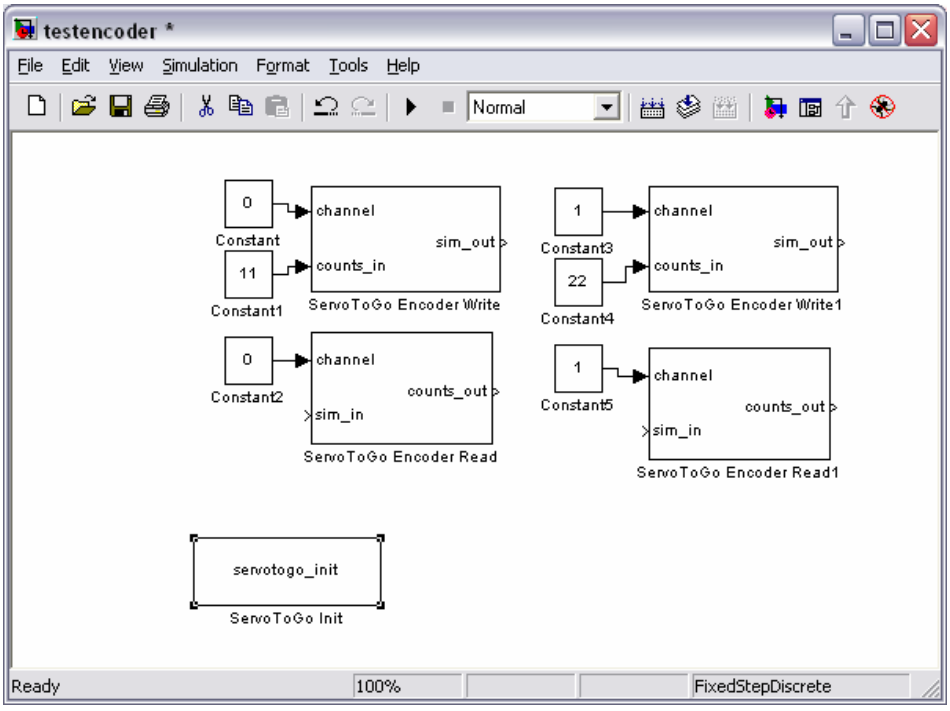
Models were created to test the functionality of each part of the system. These models are included below for reference.



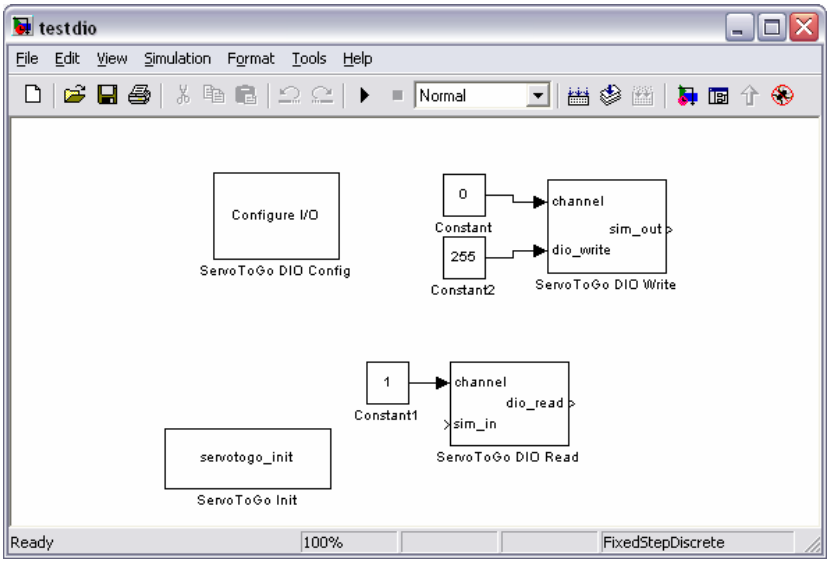
testadcs.mdl



testdacs.mdl



testencoder.mdl


testdio
File Edit View Simulation Format Tools Help
Normal
Configure I/O
ServoToGo DIO Config
0
Constant
255
Constant2
channel
sim_out
dio_write
ServoToGo DIO Write
1
Constant1
channel
dio_read
sim_in
ServoToGo DIO Read
servotogo_init
ServoToGo Init
Ready
100%
FixedStepDiscrete

testdio.mdl