Development of a Real-Time Operator Training Simulator for Falling Film Evaporators

A thesis presented in partial fulfilment of the requirements
for the degree of
Master of Technology
at
Institute of Technology and Engineering
Massey University
Palmerston North
New Zealand

Submitted by
Shane Goodwin BTech (Inf. Eng Hon's)
April 1999

Supervised by
Dr. Huub Bakker
Dr. Clive Marsh

Acknowledgements

My thanks to my primary supervisor Huub Bakker for his support and guidance, and his understanding of my highly non-linear work habits. Without his motivation and direction, the work you hold in your hands would not be there.

Thanks also to my second supervisor Clive Marsh for trying to initiate me into the intricacies of evaporators and evaporation. Better luck next time.

Thanks also to James Winchester for his efforts in evaporation education and for allowing me to dismember his pristine evaporator model in order to produce a simulator. I'm sure it was a painful process to watch.

Thanks also to the Foundation for Research in Science and Technology for their sponsorship through the Graduate Research in Industry Fellowship programme.

My thanks to the staff at Kiwi Dairies Hawera, in particular Phil O'Malley for his guidance as my supervisor at Kiwi. Thanks also to Hong Chen for his input, curiosity and for his patience in answering my many questions regarding both evaporators and the Chinese language, philosophy and culture. Xiè xiè.

Thanks also to Paul Milliken for the delivery of cookies in times of need.

And last but not least, my thanks to Maria for all her support, pickups, put downs, bribery, blackmail and general fun and fine company throughout this project. Hawera would have been a much more dismal place without her companionship.

You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat.

Albert Einstein

Table of Contents

Table of Figures

Author Publications

The follow publication was prepared during research for this thesis:

S Goodwin, H Bakker, C Marsh "Development of a Real-time Operator Training Simulator for Falling-Film Evaporators", *Proceedings of Chemeca98*, University of Queensland Australia, 1998

SUMMARY

The aim of the project was to create a simulator of a falling film milk evaporator based around the normal Fix32 SCADA control system of the evaporator, in order to train operators. This aim was broken into two objectives: the development of a generic software system to create a simulator (SRTLink); and secondly the use of that software to create the evaporator simulator (SimEvap) for operator training.

The first objective was achieved by initially laying out a specification for the software. A design of the software was then made, based around the specifications. This design was implemented and then tested to ensure that the initial specifications were met. The final software program is a package of two complimentary elements; the first a set of communication drivers for reading and writing values from a Simulink model of a process into a Fix32 control system, and the second a real-time manager for the model and its embedded communication drivers.

The development of SimEvap followed, again based around a process of laying a specification, production of a design of the simulator (based around the structure of SRTLink) and finally implementation and testing. The resultant piece of software was an executable file embodying the evaporator model (with the communication drivers embedded, as per the SRTLink structure) controlled by the real-time manager. When the generated application is run in conjunction with the Fix32 control system, a real-time falling film evaporator simulator for operator training, named SimEvap, is born.

The primary aim of the project, the construction of an evaporator simulator for operator training, was met via the achievement of the two objectives – the construction of a generic software system followed by the use of that system to create SimEvap. The constructed simulator is very similar, both visually and operationally to the real plant and is being used to train both new and experienced operators.

INTRODUCTION

The Chinese say, "Experience is the only teacher", while an old Western proverb is "Practice makes perfect". In either form, the notion is identical – the more often a task is performed, or a problem confronted, the better it is handled.

In the arena of process control involving human operators, the effects of a mistake can be extremely costly to the company. One way to reduce the frequency and severity of mistakes by an operator is to provide training, ideally on a simulation of the process rather than the process itself. This allows the operator to make mistakes and hopefully learn from them, without the company having to lose revenue, lower efficiency, or produce waste product.

An operator normally controls a process by changing the settings in a piece of software, which in turn affects the operation of the plant. However, that control software is operating on a machine which is also perfect for modelling and calculating the responses of the process in question. Software to control a process is readily available in a multitude of incarnations and so are packages designed to create models of a process. Used in conjunction, a model-based simulator can be created in order to train operators.

One obstacle to creating such a simulator is the ability (or software) to convey results and changes from the modelling software to the control software; the first objective of this project was to remedy that lack. A small piece of software was written with the overall aim to allow the construction of a real-time simulator of a process plant based on a simulation model and using the existing control system of the plant. The purpose of this simulator is to train operators without the expense of errors being made on the real plant. This led to two main tasks for the software.

The primary task of the software is to provide a communication interface between a model of a process (developed in Simulink) and a SCADA system concerned with the controlling the real world process (Fix32). The software must be structured in such a fashion that it can be "plugged in" to a model of any process, and can be configured to work with an existing control system based around Fix32.

The secondary task of the software is to run the process model in real-time. This allows for the construction of a real-time simulator based on the current control system and the process model. However, this places a restriction upon the software produced – both calculation of the simulation and communication of data must be fast enough to be handled in real-time.

With these objectives for the software in mind, a specification was developed and design, implementation and testing of the software carried out. Once created, the software was used to construct a real-time simulator called SimEvap for use by Kiwi Dairies for the purposes of operator training. The simulator models a falling film evaporator used in the production of milk powder, running it in real-time and presenting the results to an operator through the standard SCADA control screens developed for the real evaporator. With the help of an experienced operator, a neophyte can be taught how to control an evaporator.

The following thesis is broken into two main parts. The first concerns the design and development of the software as a generic platform for the creation of model based real-time simulators. The second part covers the construction of a simulator for Kiwi Dairies using the developed software. Following this are conclusions and recommendations and limitations and future developments. Both of these are discussed with respect to both the generic software and to the developed simulator.

Part I: THE FIX32/ SIMULINK SOFTWARE (SRTLINK)

1 Background

In which we take a quick tour of SCADA packages in general and Fix32 in particular, cover the synthesis of Simulink, the Real-time Workshop (RTW) and Visual C, and give an overview of the testing methodology used to develop the software.

1.1 Introduction to SCADA

SCADA (Supervisory Control and Data Acquisition) refers to a class of control systems that monitor and control a process with closed loop feedback from sensors and actuators in the process itself (Figure 1). Fix32 is an example of a multitasking SCADA system, with separate tasks for sensing, controlling and recording a process.

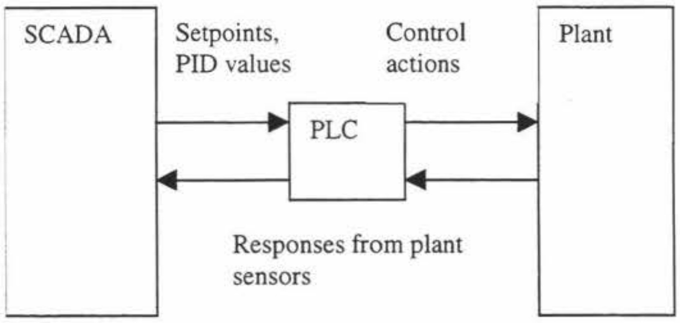


Figure 1 Standard SCADA control system setup

SCADA systems are in general computer based, rather than PLC (Programmable Logic Controller) based, although PLC's are often used as an interface between the SCADA control system and the plant. SCADA systems are designed to be monitored by operators who are responsible for setting the operating conditions and for overseeing the process as a whole to produce the desired output. The SCADA package, with the operator-chosen setpoints, is responsible for the control of the system in segments and hence the control of the system as a whole. This autonomous control is normally achieved with PI or PID control loops. These control loops can be implemented in the SCADA system itself, but more commonly PLC's are connected to the SCADA system and are responsible for the mechanics of the control using the setpoints and PID values set in the SCADA system.

1.2 Simulators and Simulation

In one form or another, simulations either have been or are being carried out on almost every process used in an industrial setting. This can range from the simple "we turn this up, that goes down" flowcharts and personal understanding of a process, to a quantitative mathematical model covering all aspects of the process under consideration to a set degree of precision.

Development of a simulation for the purposes of operator training seems to be scarce, at least outside of confidential business projects and military flight simulators. A literature review only uncovered one such project published in detail[1,2,3], although the project was only applicable by analogy to this thesis. Others were found, but were either unobtainable[4] or inapplicable to this thesis[5,6,7,8]. It was however expected that literature with any direct application would be scarce due to the tightly focused aims and methodology of the project. Work in this area may have been published in trade journals as well; however they were not part of the literature review (trade journals do not discuss a project on a technical level).

It has only become possible to both simulate and control a process on a single processor in recent years, due to the heavy demands made by both applications. The ability to create links between pieces of software running on a single processor is also relatively new. Because of these recent advances, the idea of creating a simulator by modelling the process, and then using the control system as normally used to control the modelled process is only now practical.

In brief, this means that work carried out in this area is still limited and that there has been no work done (or at least published) concerning the connection of simulation models and control systems to form a simulator for operator training. There is one exception to this; the foundation work done for this project[9,10].

1.3 Fix32 Product Description

Fix32 is one of the more common SCADA packages in the process industry. Produced by Intellution (St. Louis, Missouri), the software runs under either a Windows95 or

WindowsNT operating system. Fix32 is based around a control database that stores and manipulates the values from sensors in the plant through blocks in the database (Figure 2). A single block can refer to a particular sensor in the plant or to a calculation being performed on a value. Chains of blocks are built up consisting of sensors and calculations to monitor, and optionally, control a given part of the process.

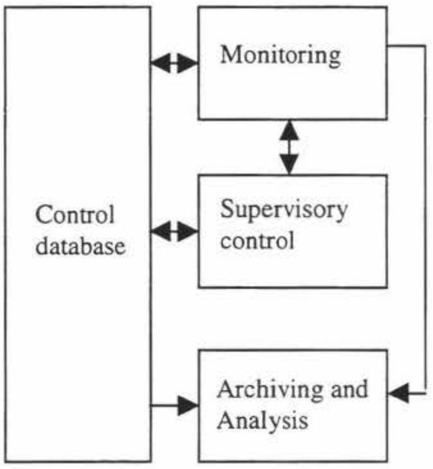


Figure 2 Internal structure of Fix32, configured as a SCADA node

A Fix32 control system is also scaleable; it can be distributed over a network. This is reflected in the structure of the control database addressing system. Any given block in the database is referred to via a three-part address called the NTF (Node Tag Field), in the form of "node:tag.field". By addressing values in this form, it is irrelevant to the Fix32 system whether the control database is local or not; both are treated in the same fashion. Benefits of this approach include easy extension of the control system as a whole (either by adding to the database on existing nodes, or by adding new nodes) and the ability to view any part of the plant (subject to the Fix32 security system) from any node in the control system[11].

1.3.1 Easy Database Access (EDA)

Easy Database Access (EDA) is both an access protocol for Fix32 and a static library of functions that can be used to access a Fix32 control database from applications written in Visual C/C++ or Visual Basic. All of the comments made with respect to EDA in this thesis use the Visual C prototypes and specifics.

EDA is designed to allow the development of external applications by third parties for use with the Fix32 system[11]. EDA allows for both the alteration of the block structure within the control database, and operations upon the values currently contained within the database. For EDA to function Fix32 must be running, either locally or on a computer that is accessible through a network.

The EDA structure is based around EDA groups. An EDA group is a list of NTF's that an operation, such as a read or a write, is performed upon simultaneously. An empty group is created with the use of *eda_define_group(n)* and NTF's are then added to the group with *eda_define_ntf.* Once the group has been filled, *eda_read* and *eda_ write* are used to send and receive data from Fix32[12].

All of the routines mentioned above, and other subsidiary routines such as *eda_get_error, eda_lookup* and *eda_delete_group* are contained in the EDA static library. Inclusion of the C/C++ EDA header files in the source, and the addition of the library in the compile options make the data in Fix32 available to a custom piece of software.

1.4 Simulink/RTW/Visual C

Simulink is a software package from Mathworks (Natick, Massachusetts) used to model a process as a flow diagram, producing a graphical representation of the flows and transformations of information in the process[13]. The Real-time Workshop (RTW) is an add-on to Simulink that will convert a Simulink model into highly optimised C code[14]. Finally, the RTW can use a C compiler such as Visual C to compile the generated code into a standalone application. The three programs in conjunction allow the visual development of a model for a process, leading to an application embodying this representation.

The RTW also can create applications by allowing support code to be included in the compilation of the model. Templates are supplied with Matlab and are designed for modification by the user to create applications with the required features. For example, the templates can be modified to carry out data logging of the model or, as in the case of this project, the management of the model to execute in real-time. It is important to note that the support code does not modify the dynamics or connections within the compiled model; indeed it cannot do so. It is designed to create an "operating system" from within which the model can be solved, and the solution then retrieved from the model for use by the created application.

1.5 Description of Relevant Simulink Blocks

Two types of blocks are particularly relevant to this thesis – each type consisting of a complimentary pair. The first type are the From/Goto blocks[15]; the second type are the multiplex/demultiplex pair[15].

From and Goto blocks are used in Simulink to connect sections of a model that are widely separated. Logically they are equivalent to simply connecting the output port of one block to the input port of the next, as shown in Figure 3.

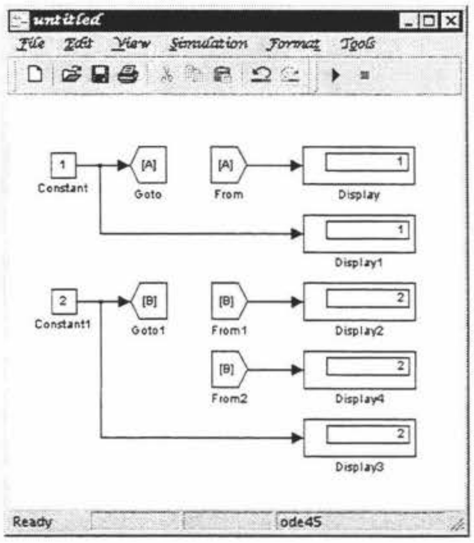


Figure 3 Example of From/Goto block pair in Simulink

From and Goto blocks each have a label; in Figure 3 the labels used are "A" and "B", as shown in the blocks themselves. These labels denote which From/Goto blocks are paired within the model. It is also possible to create multiple From blocks with the same label; this equates to multiple blocks being connected to the output of a single block. This can also be seen in Figure 3, with respect to the "B" label.

It is not possible to create multiple Goto blocks with the same label; this would logically represent multiple outputs being fed into a single input.

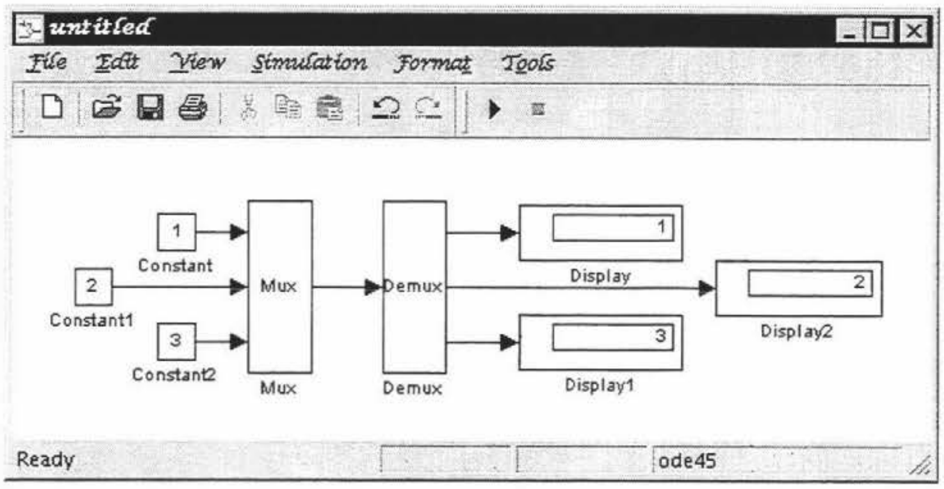


Figure 4 Example of a multiplex/demultiplex block pair in Simulink

As can be seen in Figure 4, multiplex blocks are used to combine multiple values into a vector. This vector can then be treated as a single value and passed around the model by block connections, or through From/Goto blocks. Demultiplex blocks reverse the process, breaking a vector into it's component values. Various blocks within Simulink either output vectors and/or require vectors as input.

The common terms for multiplex/demultiplex blocks are mux and demux respectively. These terms will be used throughout this thesis.

1.6 Testing Paradigms

The testing paradigm outlined here is white box/black box testing. Used as a pair, this form of software testing is practical, rigorous and simple to implement for small-scale software development.

White box testing is performed first and at the code level – the structure of the code needs to be known, and all of the conditional points tested. A conditional point is defined as a point in the code where there are two possible actions; which one is performed depends on a piece of data. For example:

```
if (TankLevel<=1) {
      LowAlarm=On;
      OpenInValve;
} else if (TankLevel>=10) {
      HighAlarm=On;
      OpenOutValve;
}
```

A fragment of code such as this must be tested so that each branch is executed, including the branch where neither LowAlarm nor HighAlarm is set on. This requires the code to be run with TankLevel set to a value of less than one, greater than ten, and between one and ten. Note that, if conditional statements are found nested inside other conditional statements, a combinatorial situation develops. All of the branches of the interior condition must be tested under all the conditions in the outside condition.

Black box testing does not require knowledge of the internal workings of the code – a piece of code that purports to perform a certain function is regarded as a black box. Under this definition, any piece of code that has been white box tested creates a black box. Any library functions used are inherently regarded as black boxes (there is normally no code-level access to libraries).

The testing of a black box is performed by varying the situations in which the "black box" is called. Say a function `CheckTank(i)`, supplied from a library, is used by the software. Supplying different values of `i` tests the black box. If no bugs can be found by varying `i` from valid values to invalid ones, the black box routine `CheckTank` is considered bug free.

With respect to supplied libraries of functions, black box testing is assumed to have already taken place – the developer of a piece of software using a library does not normally black box test the functions they use. In practice, this is a valid assumption to make; function libraries are normally fairly bug free after being in the public domain with users finding and reporting bugs to the developer. However if, during the white box testing of the software, an error is found that could be caused by a library function, the call to the library function is removed and the software run to see if the error persists (this is normally called deadly debugging). In this fashion, any errors generated by the "untested" libraries, within the context of the software that is using them, are checked for.

2 Development of SRTLink

In which we travel through the development of SRTLink, via the software specification, a design to meet that specification, implementation of the design, and testing of the developed implementation to ensure the specifications are met.

2.1 Specification

The software was to be used to create a model-based real-time simulator. The model of the process was to be constructed within Simulink and the control system for that process was to be constructed in Fix32. The created software must therefore be able to communicate with and use the facilities of both Simulink and Fix32.

To be of practical use, the software needed the capability to pass at least a hundred values, both from the model to the control system and from the control system to the model. This total of 200 values must also be communicated at no slower than once a second in order to create the impression of continuous operation for the trainee. The process of communication also needed to be quick enough to allow both the simulation to run and the communication to occur in real-time on a given processor (a 200Mhz Pentium, running WindowsNT).

The software also needed to be capable of keeping the communication and the simulation synchronous with real-time. It was decided that the limit of this should be 1%; that is, the simulation was allowed to lose or gain 0.01 seconds for every second simulated. Note that this figure is not with respect to overruns by the simulation software, merely with respect to the limit of accuracy required in normal operation.

The software also has to be able to handle overruns when they occur, either because of difficulty in solving the system of equations within the model or because of problems relating to the communication of values between the model and Fix32. The system to handle overruns must allow for the continuation of the simulation with as little disturbance as possible from the viewpoint of the trainee.

It was also seen as important that the software be generic, so that it could be used with any model of a process. This allows for a pre-existing model to be used for the development of a simulator, as opposed to constructing or restructuring a model to allow for the incorporation of the software. Because models in some form normally exist for most processes, this point was seen as crucial to the overall usefulness of the software.

Emphasis was also placed on ease of use for a person developing a model and producing a real-time simulator with their model. This meant some form of configurable user interface was necessary to allow the software to be set up for different situations by the user. This also entailed making the method of connecting the model and the software as streamlined and flexible as possible.

2.2 Design

Given the previous specification, it was decided to split the software into two components, each with respective tasks. The first component would be in the form of drivers embedded in the Simulink model itself, responsible for communicating values between the model and Fix32. Two separate drivers were envisioned: one concerned with the flow of information from Fix32 to the simulation model (FromFix) and the other with the flow of information from the model to Fix32 (ToFix) (Figure 5).

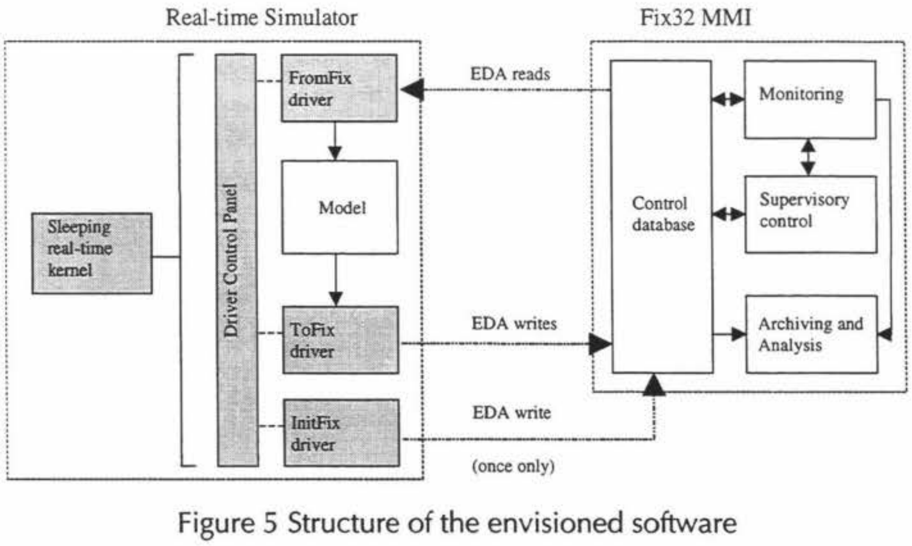


Figure 5 Structure of the envisioned software

In order to make it as simple as possible to connect the drivers into a given model, visually and operationally they need to be in the form of normal blocks within the Simulink environment (Figure 6). The blocks are then dropped into the model in the same fashion that any other block would be, and connections between the drivers and the model made via From/Goto blocks. By keeping the drivers along the lines of standard Simulink blocks, it was hoped that a developer familiar with the normal practices in creating a Simulink model would have little difficulty integrating the communication drivers into a model.

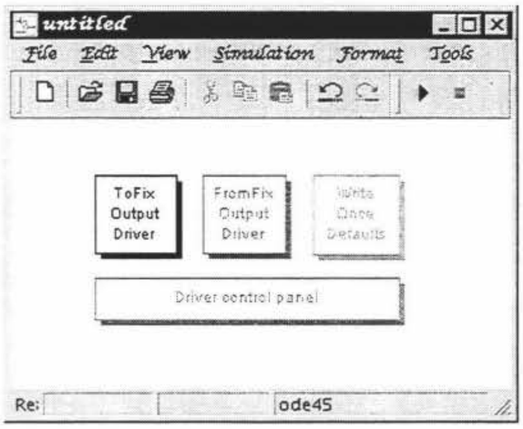


Figure 6 Screen shot of the drivers and the Driver Control Panel

The task of the second component was to run the simulation model in real-time, including the embedded drivers. This component monitors the length of time taken to calculate a time step in the simulation, and then put the simulation model to "sleep" while an appropriate amount of real time elapses to synchronise the simulation with real time. In this fashion the simulation is kept synchronised with reality to within the 1% limit laid out in the software specification. Where this component fits into the overall structure can be seen in Figure 5.

A Driver Control Panel (DCP) was also envisioned to further increase the usability of the software from a developer's point of view. The control panel was to control the execution interval of the drivers and to allow the user to enable and disable them in order to further develop the model of the process, independently of the Fix32 control system.

2.3 Implementation

2.3.1 S-Function I/O drivers

S-functions are specialised Simulink blocks that are used to run user-written code inside a model[13]. The code for S-functions can be either an M-file or C code, which has been compiled into a dynamically linked library (DLL). A library that has been compiled from an M-file is referred to as a MEX file, whereas a library generated from C code is referred to as a CMEX file. The I/O drivers are examples of CMEX files, although they are stored on the disk as standard 32 bit Windows DLL's, with the DLL extension.

For a given CMEX file, Simulink expects a number of entrypoints into the library with specific names[13]. These functions are required in order to interface the S-function to the inputs and the outputs within the model, and to allow Simulink to control the activation of the S-function for the purposes of initialising and running the code within the structure of the model. The calling sequence (and entrypoints) of the driver CMEX functions can be seen in Figure 7. The entrypoints outlined in bold are required by Simulink, while other functions (such as *mdlStart*) are not actually compulsory for Simulink to execute the block.

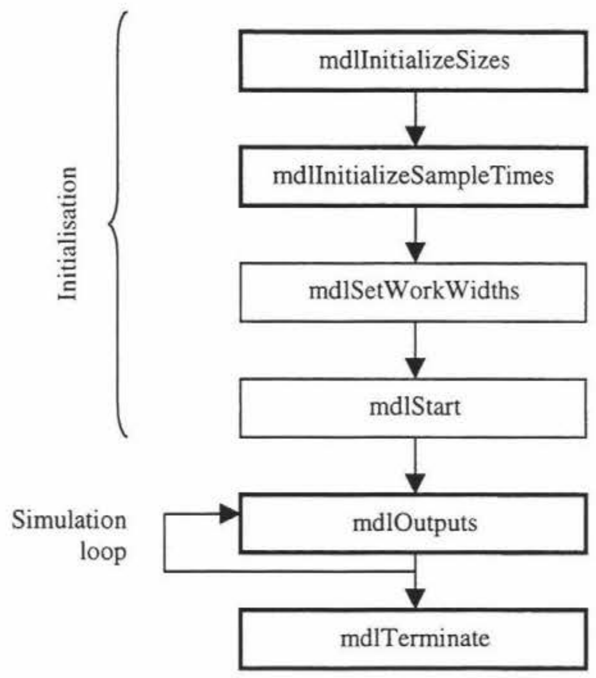


Figure 7 Calling sequence and entrypoints of driver CMEX functions

Each of the entrypoints in the CMEX file are essentially a function in the source code, and each are used for different purposes. *mdlInitalizeSizes* is to set the size of the input and output vectors, to set the user configurable options for the S-function and to set aside memory for the internal work buffers of the S-function. *mdlInitalizeSampleTimes* is to set up the sample timing of the S-function block with respect to the rest of the simulation blocks. *mdlSetWorkWidths* is to set the size of the user-accessible work vectors for the S-function. *mdlStart* is run only once, being in the initialisation phase of the simulation, and is used for custom code that only needs to be executed once. *mdlOutputs*, on the other hand, is run in every simulation time step and again contains custom code. These two functions together should contain the code to perform the task required of the S-function. And finally, *mdlTerminate* is executed when the simulation finishes in order to release memory used by the S-function. Note that there are other *mdlXXX* functions with varying purposes that are not shown here and that only *mdlInitalizeSizes, mdlInitializeSampleTimes, mdlOutputs* and *mdlTerminate* are compulsory; all of the other *mdlXXX* functions are optional[13].

In the case of the I/O drivers, the *mdlStart* routine was used to configure the driver upon the initialisation of the model. This involves reading in from a disk file a list of tags to be read or written in each communication interval and setting these tags up as an EDA group, ready to be used. Once initialised in this fashion, reading or writing values through a group is performed as a single function call. This takes place in the *mdlOutputs* routine within the driver. Simulink calls this function once on each time interval and the communication takes place if the time interval falls on a communication interval. The size of a communication interval is set in the Driver Control Panel within the model.

2.3.1.1 FromFix

FromFix is the communication driver responsible for getting values from the Fix32 system and making them available for use within the model. Commonly, these values would be setpoints for control loops set by the operator and the PID values for the control loops.

The general structure of the driver as it is in Simulink is shown in Figure 8. The tag/fields for each of the channels in the driver are read in from a configuration file named "FromFix.cfg" (a channel is defined as the path for a single value, either from or to Fix32). The values read in from these tag/fields by the driver are made available on a dynamically-sized vector, which is fed into a demux block. Each of the of the individual values in the vector are then sent into a Goto block with a unique name in the form of "F001", "F002", and so on. A From block paired with each Goto block (ie. containing the same identifier tag) is then used within the model to supply the value. Note that multiple From blocks can be paired with a Goto block and each From block will supply the same value.

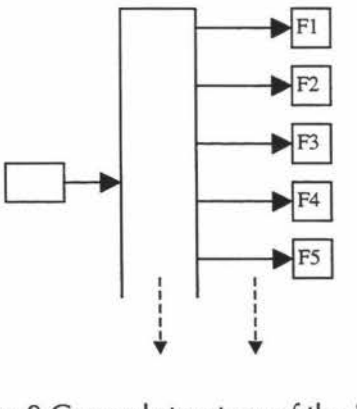


Figure 8 General structure of the FromFix driver

With this structure, integrating the FromFix driver with an existing model is quite straightforward and doesn't require restructuring of the model. The FromFix driver subsystem itself can be dropped anywhere in the model (although typically this will be at the top level of the Simulink model) and anywhere within the model where a value is required a From block with the desired identifier can be inserted.

To make the driver easier to place into a model, a set of paired Goto/From blocks are laid out within the driver subsystem and all of the blocks are colour coded magenta. Laying out the Goto/From blocks in this fashion is for two reasons. The first one is that whether or not a particular channel is used in the model, there is always at least a single From block for each Goto. This prevents Simulink from warning the user about unmatched Goto blocks when the model is run. The second is by already having the From blocks available within the model they can just be copied into the desired locations within the model from the driver subsystem, rather than the user having to create them from scratch. A picture of part of the FromFix driver as it appears in Simulink can be seen in Figure 9.

FromFix Input Driver
File Edit View Simulation Format Tools
FC01 FC02 FC03 FC04 FC05 FC06 FC07 FC08 FC09 FC10 FC11 FC12 FC13
tag.field
Demux block
Goto block array
Text labels for documentation
From block array (to be copied into model)

Figure 9 Screen shot of the FromFix driver as it appears in Simulink

To enhance the usefulness of the driver, a further feature was added. An array of values can be seen in Figure 10, as part of the FromFix driver. These values are in line with the tags that they represent. If the driver is disabled and the Sim/EDA switch (see below) turned to "Sim" the values in Simulink are used instead of those from Fix32. This allows for easy development of the model after the drivers have been inserted by dissociating the input from Fix32.

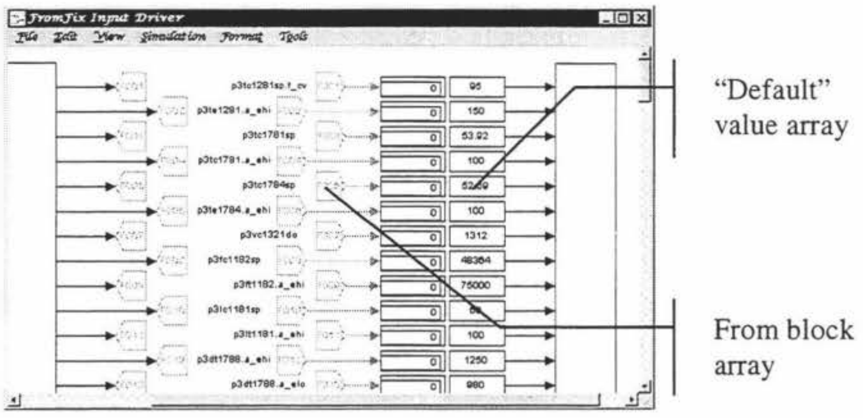


Figure 10 FromFix driver array of "default" values

2.3.1.2 ToFix

ToFix is responsible for supplying values to Fix32 from the model. These values are normally all of the process responses that the control system would expect to receive from the plant via the PLC's. This of course assumes that the model is able to provide all the necessary information (i.e. that all parts of the process are modelled).

With respect to the code, ToFix is very similar to FromFix – in fact there are only two differences. The first is the name of the configuration file that is read in ("ToFix.cfg" as opposed to "FromFix.cfg") and the second is that *mdlOutputs* writes data via the EDA group, rather than reading it.

The structure of ToFix within Simulink is essentially a reflection of FromFix and can be seen in Figure 11. Rather than the driver feeding a demux block (which in turn feeds the Goto/From pairs for the driver) a set of Goto/From blocks feed values from the model into a mux block (which in turn feeds the driver). The incoming trunk of values is matched with the tags read in from the configuration file and the values written to the control system via the EDA functions.

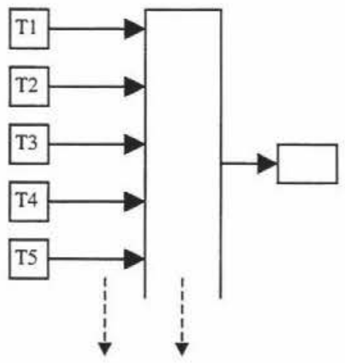


Figure 11 General structure of the ToFix driver

Unlike FromFix however, once the Goto end of one of the inputs into the driver is inserted into the model, its twin must be removed from the driver. If it is not removed, then Simulink will be unable to run the model – the presence of multiple Goto blocks with the same label is not allowed.

The blocks within the ToFix driver are colour-coded blue. This is to allow the user to easily see which driver is being consulted at a point in the model. A picture of part of the ToFix driver as it appears in Simulink can be seen in Figure 12.

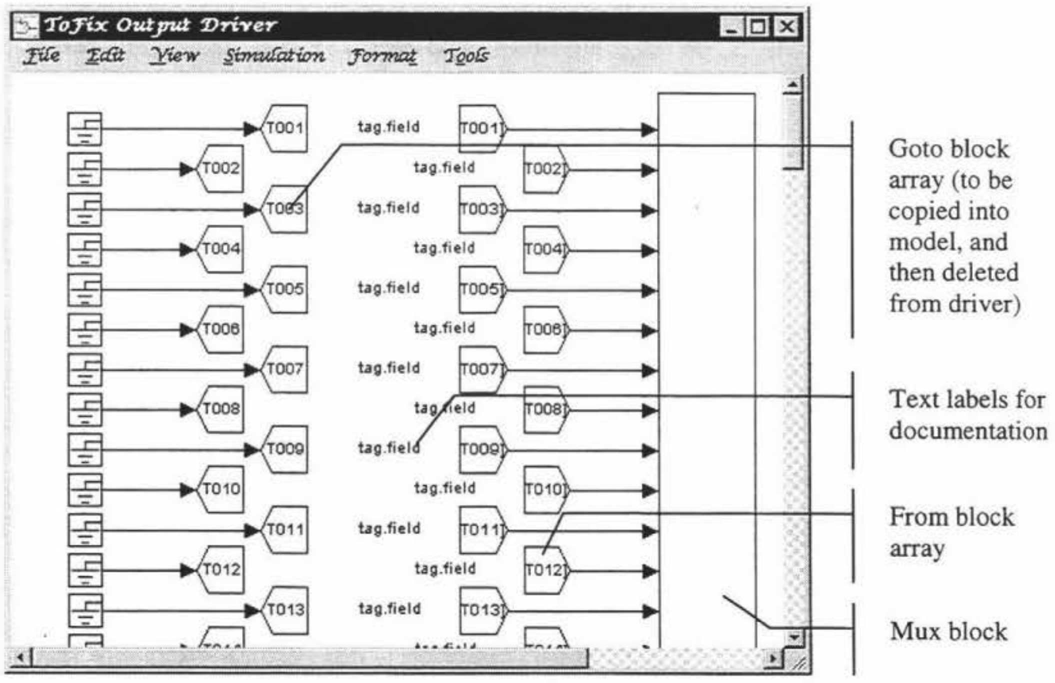

ToFix Output Driver
File Edit View Simulation Format Tools
T001
T002
T003
T004
T005
T006
T007
T008
T009
T010
T011
T012
T013
tag.field
Goto block array (to be copied into model, and then deleted from driver)
Text labels for documentation
From block array
Mux block

Figure 12 Screen shot of the ToFix driver as it appears in Simulink

2.3.1.3 InitFix

The purpose of InitFix is to configure Fix32, ready for the beginning of the simulation. The reason this is necessary can be seen clearly by analogy; if you mix up the ingredients of a cake, put it into a cake tin and put it in the oven, you wouldn't expect it to rise if you hadn't set the oven to 180°C, would you?

The InitFix code is in structure somewhat different to the other two I/O drivers, because once the simulation is running it does not perform any tasks. The *mdlOutputs* routine within the driver is empty, and the code executed within *mdlStart* is also somewhat simpler than the code found in its cousins, because no communication of EDA groups is necessary between *mdlStart* and *mdlOutputs*.

When the driver is initialised by the simulation, it reads in from a text file (normally called "InitFix.cfg") a list of tags and values, a pair of which occur on each line in the form "1 = tag.field,*x*". Each of the values is written out to its respective tag and field by InitFix, and then made available on the block outputs within Simulink in the event that the model also needs it.

The structure of the driver within Simulink is similar to FromFix and ToFix, although the values that are available are static. However, normally these values would not be used from the block outputs of the InitFix driver – they would be retrieved via the FromFix driver. This allows for the default start-up value set by the InitFix driver to be later changed by the operator. An example of this situation would be a setpoint for a control loop; the loop is set to its initial desired condition by InitFix within Fix32 and the model then uses FromFix to retrieve that value at each communication interval. This allows the user to alter the setpoint later in the simulation and have the model reflect the change.

2.3.2 Real-time Kernel – srt_main.c

The SRTLink OS is based on the "grt_main.c" template supplied with the RTW, modified to simulate a compiled model in real-time. In addition to this, the OS can perform data logging of the time taken for each calculated step within the simulation and will handle any overruns of the model as well. The configuration options for the SRTLink OS are set within a configuration file named "SRT.cfg" read each time the application is run. For a list of the configuration file options, refer to the SRT Configuration File Options section in Appendix C: SRT User Manual.

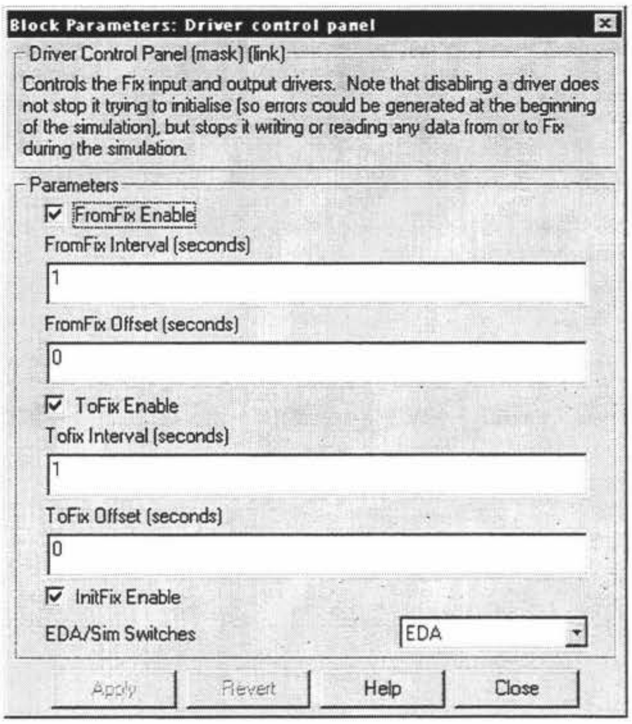
The calculations performed by the OS to run the model in real-time are based on the sample time read from the compiled model. The model is run for one step (whether that is 0.1 seconds of simulation time or 10 seconds of simulation time) and the time taken to perform the calculation measured by the OS. Assuming that the calculation of the model time step has been performed within the amount of real-time, the OS then puts the simulation to "sleep" until the required amount of real-time has elapsed. In this fashion the model is kept synchronised with real-time.

Overruns are handled by doing exactly nothing. The reason for taking this approach is the realisation that nothing needs to be done. The next time step is calculated as usual. By handling overruns like this, the simulation is kept running, albeit with a small (and unknown) slip in time. From the perspective of the trainee running the simulator, there is no real interruption of the process; the control screens just don't change for a slightly longer interval than usual.

Any overruns are, however, noted and written into the log file by the SRTLink OS. The nominal name of the log file is "Logfile.txt" and can be changed in the "SRT.cfg" file.

2.3.3 Driver Control Panel (DCP)

The driver control panel, like all of the I/O drivers, just needs to be dropped into the top level of a model for it to be used by the system. Unlike the drivers however, no modifications by the user are required once it is in the model. Clicking on the Driver Control Panel (DCP) subsystem block brings up a mask allowing the user to select the intervals for communication, to enable or disable the drivers and to change the setting of the EDA/Sim switch (Figure 13).



Block Parameters: Driver control panel

Driver Control Panel (mask) (link)

Controls the Fix input and output drivers. Note that disabling a driver does not stop it trying to initialise (so errors could be generated at the beginning of the simulation), but stops it writing or reading any data from or to Fix during the simulation.

Parameters

☑ FromFix Enable

FromFix Interval (seconds): 1

FromFix Offset (seconds): 0

☑ ToFix Enable

Tofix Interval (seconds): 1

ToFix Offset (seconds): 0

☑ InitFix Enable

EDA/Sim Switches: EDA

Figure 13 Driver Control Panel (DCP) mask

The major task of the Driver Control Panel is at a higher remove from the user, however; it is responsible for the timing and triggering of the drivers, using the intervals supplied by the user. The timing mechanism is implemented within Simulink in the DCP subsystem (Figure 14). The system, by performing a remainder check at each time step, decides whether the current step is a communication step or not. This calculation is carried out independently for both FromFix and ToFix (InitFix is not used after the initialisation phase of the simulation, so no timing trigger is necessary).

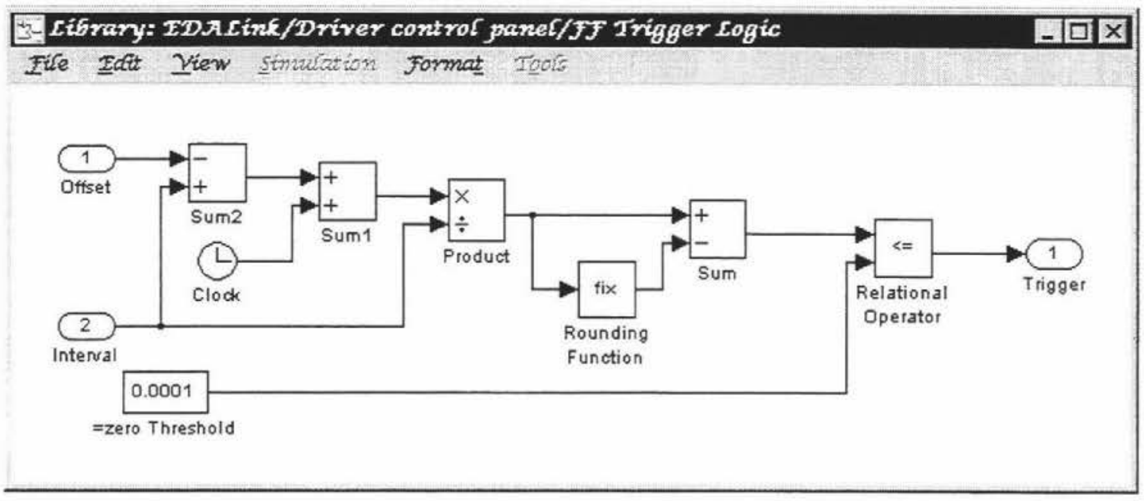


Figure 14 Timing system for the drivers, within the DCP

2.3.4 EDA/Sim switch

Once the drivers are inserted into a model and connections between the SCADA system and the model are made with From/Goto blocks associated with the drivers, the model is reliant on Fix32 and the related control database being active whenever it is run. This was seen as unsatisfactory if further development of the model was required (as it often is). Ideally, the user should be able to turn the drivers off, effectively removing the link. This introduced a second problem however – once inactive, the drivers will not supply values to the model, such as setpoints for the control loops. This meant that the model could not be run, as it did not have all of the necessary information to simulate the operation of the process.

The EDA/Sim switch was designed to get around this problem, allowing the continual development of the model with the drivers in it but inactive. In effect, it creates a parallel set of values that can be set and used within Simulink whenever necessary. By adding a control setting to the DCP, the user is given the flexibility to develop the model with the drivers in it, but with or without the Fix32 system present and running.

An example of the use of the EDA/Sim switch can be seen in Figure 15. The model is designed to calculate the number of seconds in a given number of hours. Normally this conversion would be performed on a value being retrieved from Fix32 via the FromFix driver (through the F001 From block at the left of the model). However, instead of just connecting the From block to the input of the multiplier block, it is first passed through an EDA/Sim switch. The second input of the switch, labelled "Sim", is fed a value from a constant block (set to 1.5). Thus, when the model is compiled or the developer wants to use inputs from the control system, the EDA/Sim switch in the DCP is set to *EDA*; but to allow the development of the model without the control system present, the switch is set to *Sim*.

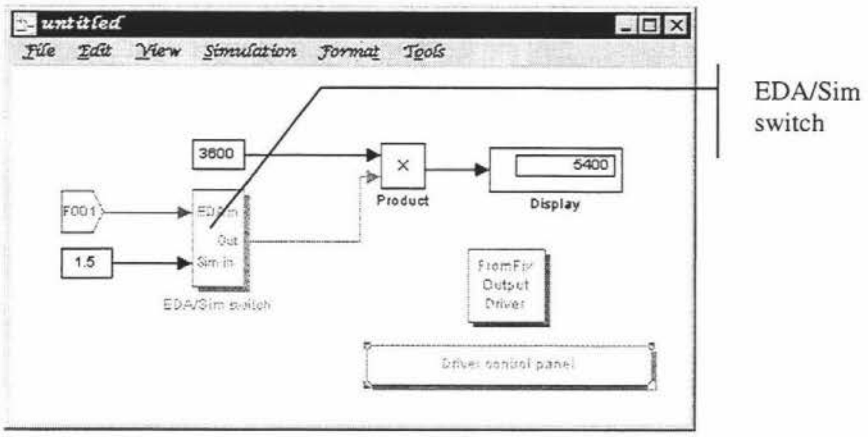


Figure 15 Example of an EDA/Sim switch in use

By using EDA/Sim switches in this fashion throughout the model, it is possible to disassociate the drivers from the rest of the model with the flick of a toggle. Development of the model can then proceed without the need of Fix32.

2.3.5 Fix32 Configuration

Before Fix32 will accept data through EDA, as opposed to through any connected PLC's, the blocks within the control database that are to receive the data must be set up correctly. This is to ensure that stray EDA calls cannot alter a database that is being used for the control of a process[12] (this restriction does not apply to reading data from blocks via EDA).

Three changes are necessary, all of which should be applied globally to the database (assuming a pre-existing database).

1 The "Init. Auto/Manl" field for all the blocks should be changed from AUTO to MANL. This is the main change that is necessary; blocks that are in automatic

mode will not accept input from any source other than the Scan, Alarm and Control task (SAC)[16].

2 Any occurrences of an "E" in the "Scantime" field of a block should be altered to *x* seconds; a value of 1 second is nominally reasonable. An "E" in this field will only allow the triggering of a block (and hence the execution of its associated chain) via an event generated by the SAC, such as an alarm. Using EDA to insert data into a block configured like this results in the chain remaining inactive. For example, if the block is the beginning of an alarm handler, no action is taken when an alarm value is inserted into the block via EDA because the chain attached to the block is not activated.

3 Because a node running a simulator normally will not have any PLC's attached to it, leaving the driver set to reference a PLC results in problems with an undefined Driver Image Table (DIT). To avoid this problem, all occurrences of the driver name (such as ABR, ABH, and so on) should be altered to SIM. The SIM driver is an internal driver to Fix32, and does not need nor build a DIT. Unfortunately, selecting a different input driver for a block resets the I/O address of the block to 0. This can result in multiple blocks looking in the same memory location for their current value. To circumvent this, once the I/O driver has been changed for a block it must be given a unique memory address within the range of the SIM driver (0-2000)[16].

2.4 Testing

White box and informal black box testing was chosen for this project for a number of reasons.

White box testing is easily performed, if somewhat tedious when it comes to convoluted systems of conditional responses. In the case of this project, the software to be written is quite small in comparison to the size of a fully developed application, and this meant that white box testing was not too expensive in terms of time.

White box testing is more rigorous than other bug testing methodologies. If a full sized application is being developed running to hundreds of thousands of lines of code, rather than the approximately two and a half thousand of this project, full white box testing becomes pragmatically impossible. Methodologies designed to cope with large-scale programs of necessity lose some of the rigor in order to become practical. This trade-off was not necessary for this project.

Formal and complete black box testing was not seen as strictly necessary because the library functions used within the software have been in use for some time. This means that they have been tried under many differing circumstances, and they can be assumed to be quite reliable. Because of this approach, however, when any errors were found and traced to the vicinity of a library call, the library was considered suspect. This added uncertainty was offset by the use of deadly debugging as a first test (this combination of testing methodologies is quite common when widely tested libraries are being used, but have not been black box tested for a particular piece of software development).

Other bug testing methodologies are based around multiple developers and large-scale projects, and have an overhead of documentation to co-ordinate the development. Because this project involved only one person this formal overhead was not seen as necessary; standard development protocols and notes were seen as sufficient.

Part II: THE EVAPORATOR SIMULATOR (SIMEVAP)

1 Background

In which we take a look at the general background of evaporation and evaporators, their purpose and the evaporators at Kiwi Dairies, along with their control systems.

1.1 Evaporation

evaporate / I∪ ϖΘπ↔∩ρεIτ / *v.* 1 turn from solid or liquid into vapour. 2 lose or cause to lose moisture as vapour. evaporation / √ ∪ρεIΣ(ε)ν / *n.*

The dictionary definition[17] of evaporation covers both the process and the desired outcome of the process. The reason *why* the process is performed is not given and, indeed, could not be in a concise form. Evaporation of water is used in hundreds of applications, from processing food (drying or concentrating) to purification (distillation and desalination) to the production of power (steam turbines and geothermal power stations).

The application we are concerned with here is the evaporation of water from milk in order to produce milk powder. One common method of concentrating milk is with a falling film evaporator[18]. Milk is allowed to fall down the interior of a tube in a thin film, under vacuum. Heat is supplied via steam condensing on the exterior surface of the tube (Figure 16).

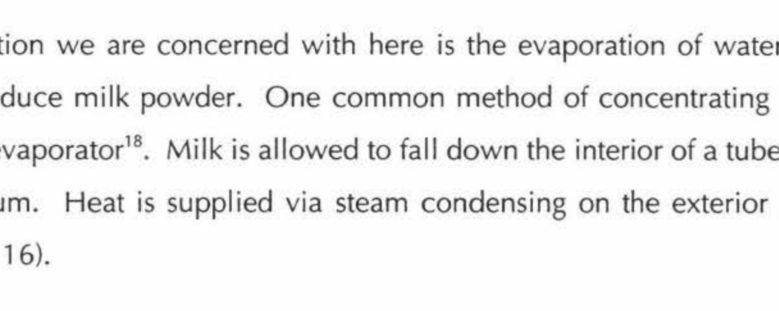


Figure 16 Cross-section of a tube in a falling film evaporator

Because the milk is under vacuum conditions, it only needs heating to ~66° in order to evaporate the water. Also, the time taken for a section of the film to fall the length of the tube is constant; this is important because it ensures that all the milk falling down the tube is heated equally.

As more water is removed however, the film becomes more viscous which can result in build-ups of milk becoming burnt onto the side of the tube, resulting in fouling. In general, the limit for concentrating milk in a falling film evaporator is ~50% milk solids, although this varies by up to 2% with the kind of milk being concentrated (milk enters the evaporator at ~12% total solids). If the milk is concentrated any further, fouling begins to occur.

To remove the rest of the moisture in the milk, a spray dryer is normally used, leaving the final product above 99% total solids.

1.2 Evaporators at Kiwi Dairies

Kiwi Dairies has five falling film evaporators installed for the purposes of manufacturing milk powder. They range in size from 2 tonne an hour (Powder 1) up to 20 tonne an hour (Powder 5). We are concerned with the Powder 3 evaporator for the construction of the simulator. This means that the control screens used are those for Powder 3 and the model is designed to reflect the physical size and properties of Powder 3.

The Powder 3 evaporator has a total of 8 passes, 5 with MVR (Mechanical Vapour Recompression), and 3 with TVR (Thermal Vapour Recompression). 50,000 litres/hour of milk solution enters the evaporator at approximately 12.5% total milk solids, with an output from the evaporator in the region of 14, 000 litres/hour at ~50% total milk solids.

The MVR operates by recompressing the steam evaporated from the milk with a compressor, hence heating it up before injecting it into the shell to evaporate more water from the milk. The TVR works by straight injection of steam into the shell of the evaporator from an external source. An MVR is much more energy efficient, but control is difficult due to a coupling between the steam temperature and the milk temperature. The TVR, while much less efficient, is much more controllable.

1.3 Fix32 at Kiwi

Fix32 is used at Kiwi Dairies for control of the Powder 3, Powder 4 and Powder 5 plants, as well as other production plants onsite. Fix32 is installed on a distributed node network linked to Allen Bradley PLC's by an Allen Bradley Data Highway (refer to Figure 17). Because the model is based on the geometry and configuration of Powder 3, we will restrict discussion of the control system to that based around Powder 3 although the general structure is the same across all of the three powder plants.

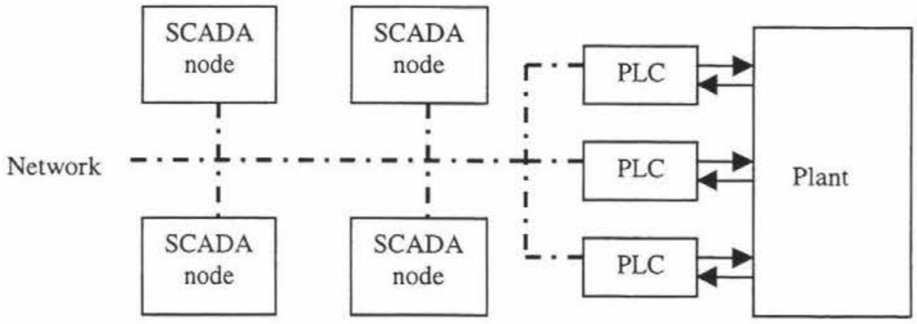


Figure 17 Fix32 network layout for controlling the powder plant.

Within the network, a few computers are configured to run Fix32 as SCADA nodes for each plant. Typically, there is a node concerned with the evaporator (called POWDER3), one concerned with the dryer (P3DRYER) and one concerned with the feed milk silos (PSILOS). Each of these nodes maintains a database updated by Fix32, nominally every second, with the values from the sensors on the plant.

These three computers are normally devoted to displaying the status of the respective plants, although any of these computers can display information from any part of any of the Powder plants. None of these viewing nodes have the access rights (or installed components) to alter the control database or control screens – they are for the use of the operators in managing the plant only.

All of the control screens are laid out in a general form as per the example given in Figure 18. The top of the screen has a menu bar concerned with major selections, such as which plant (Powder 3 evaporator, Powder 4 drier and so on), under which is a second bar for choosing which view of the plant is desired. Under that is the information, relevant to each page (shown in Figure 18 is the Evaporator A overview).

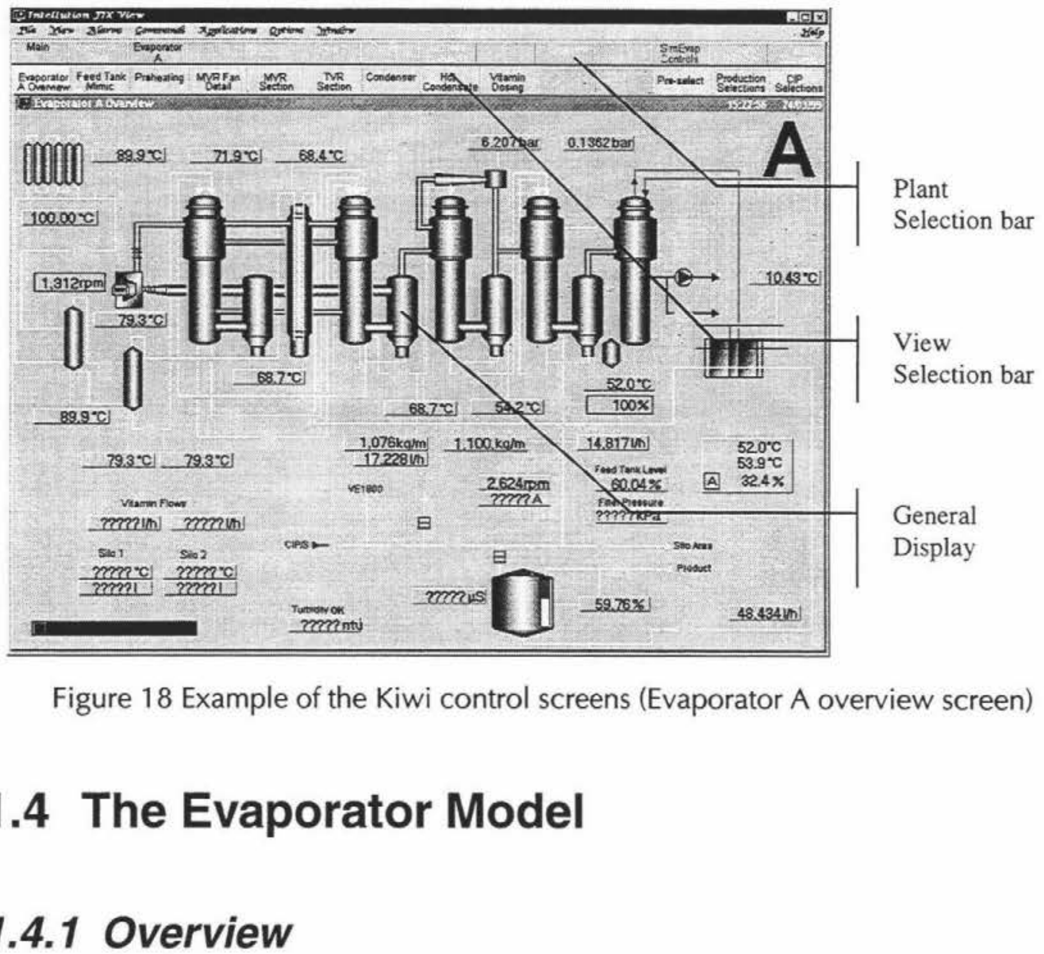


Figure 18 Example of the Kiwi control screens (Evaporator A overview screen)

1.4 The Evaporator Model

1.4.1 Overview

The simulation model[19] describes the dynamics of the multi-pass/multi-effect falling film evaporator installed as Powder 3 at the Kiwi Dairies milk powder plant. The model was originally developed by James Winchester along thermodynamic lines to provide a method of better understanding the operation of the evaporator.

The variables input to the model are; the feed flow, the feed temperature, the control inputs and the feed dry matter. The model is a first principles derivation[20,21], based on mass/energy balances of 5 coupled subsystems of the evaporator; the distribution plate, the evaporator effect, the evaporator shell, the concentrated product and the condenser. These coupled subsystems are used to model each of the four sections of the evaporator shown in Figure 19. One of the major complications of the model is the falling film, which depends on distance down the tube - this is modelled with a partial differential equation, using 5th order Pade approximations for the delay induced by the residence time of the falling film.

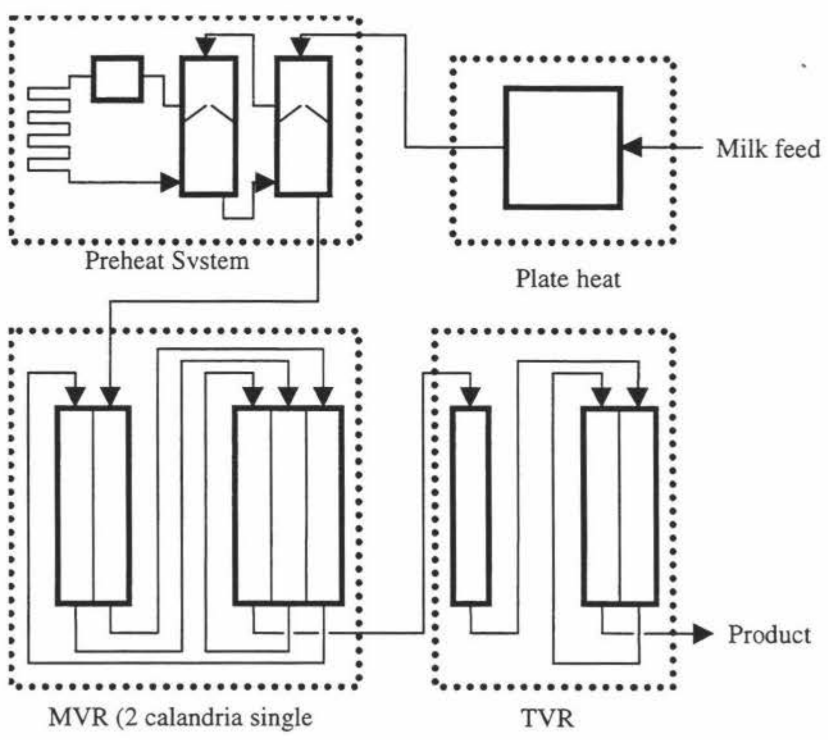


Figure 19 Evaporator model outline

The model as a whole can be expressed with as few as 10 differential equations, if simplifications are made (resulting in the loss of some accuracy). In the order of 100 differential equations are required for the complete model.

1.4.2 Restrictions on the Model

Implementing a model on any practical computer (ie. one that does not consist of infinite amounts of memory and processing power) results in some approximations becoming necessary and therefore some restrictions being placed on the simulation.

With respect to Simulink, these restrictions are in major part due to approximations caused by binary rounding. Further limits are with respect to the sample time of the model – theoretically this can be as large as desired but, as the size of the sample interval increases with respect to the speed of the model dynamics, so too does the size of cumulative errors within the simulation. Ideally, the model would be simulated at the same speed as the desired communication interval (in this case, one second) and communication would occur in every time step. This is impractical, due to the size of the errors that accumulate with a one second time step (the internal model dynamics are in terms of tenths of second or less). This means that the model was simulated with a sample time of 0.1 seconds.

2 Development of SimEvap

In which we travel through the development of SimEvap via the software specification, a design to meet that specification with the help of SRTLink, implementation of the design and testing of the developed system to ensure it meets the specifications.

2.1 Specification

The objective of this part of the project is to create a system to train operators in the control of the evaporation process on a simulator that matches the "real thing" as closely as possible, both visually and with respect to the response to operator actions.

Maintenance of the visual properties of the created simulator, in order to keep it up to date with the real control system of the process, must be as simple as possible. There must also be scope to allow the development of better responses (ie. closer to the responses of the real plant) to operator actions. This also includes any modifications made to the plant that need to be reflected in the simulator. Ideally, the development of these improvements should be independent – after all, they do not proceed synchronously in the real world either.

The developed simulator also needs to visually operate no slower than every second; ideally it will update every second on the second. This limit was picked to match the standard operating characteristics of the real control system; the update interval of the PLC's on the plant is also one second.

It was perceived that the operators who would be using this system, either as a training tool or as a "what-if" tool, would not be mindful of the workings of the simulator as a whole. This meant that some form of control system for the process simulation must be incorporated into the simulator, in as seamless a fashion as possible from the viewpoint of the operators using it.

2.2 Design

Because the developed software SRTLink is to be used to create the simulator, the design of SimEvap must be based around the structure of SRTLink. This means that a model of the evaporator process is needed and that the software must be integrated into this model.

Most of the objectives of the specification are inherent in the design and structure of SRTLink. The ease of maintenance of the visual displays to keep them up to date with the real plant is taken care of with the use of Fix32 and the real control screens being used by SRTLink as the front end of the simulator. SRTLink also acts as a buffer to create an independence between the development of the display screens and the development of the model. Finally, SRTLink allows the communication interval to be set as desired; setting it to one second matches the visual updates to those of the real Fix32 control system – the ideal solution.

Because the operators who are using the simulator are envisioned as having previous experience with Fix32, or at least having somebody on hand with such experience, the controls for the model are to be implemented as part of the Fix32 system. This equates to adding an extra page of controls relevant to the model to the Kiwi Fix32 control system.

2.3 Implementation

2.3.1 Changes made to the model

Two major modifications were made to the model; the addition of the tanks and valves, and addition of PID control blocks to the systems within the model. The original model was structured to provide information about the flows and temperatures and was not designed to reflect the physical structure and control attributes of the evaporator. For the purposes of a real-time simulator, this was not sufficient – the modelling equivalent of the PID controllers and actuators were also necessary, including the tanks and valves to be controlled.

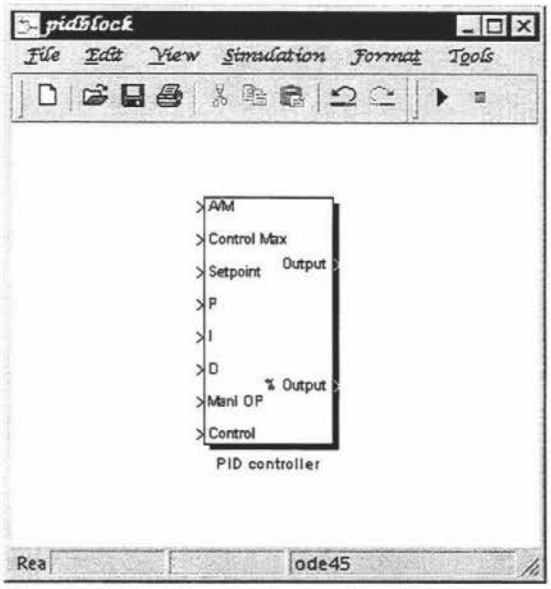


Figure 20 PID controller block at the connection level

With respect to the model, this meant constructing a PID controller within Simulink. The developed controller needed to have the same operating characteristics as those used on the plant, which meant bumpless transfer and anti-integral windup were a necessary part of the controller design. The overall structure of the PID controller must also be compatible with the switches and values that are part of the SCADA system developed by Kiwi. The block level interface of the controller can be seen in Figure 20.

Figure 21 shows the first level of the PID controller subsystem. At this level the PID controller itself can be seen and the manual/auto switch to enable or disable the controller output. The top right corner performs a difference calculation, and a normalisation of the control error to a value between zero and one for the PID controller. The controller action is also supplied as a value between zero and one, and is normally modified or scaled in some fashion to reflect the true value of the control variable before being fed back into the model.

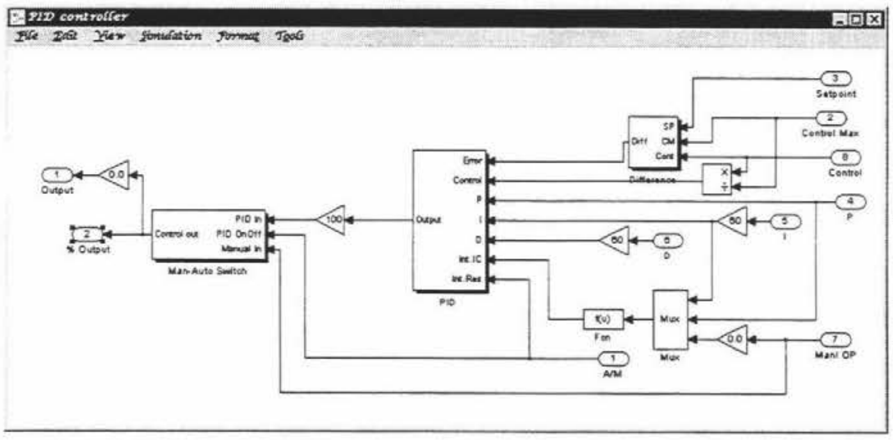


Figure 21 First subsystem level of the PID controller block

An example of a PID block in use is shown in Figure 22. This controller reflects the tank level controller on the evaporator feed tank. To the left of the PID controller can be seen inputs via the FromFix driver feeding settings into the control block. The controller action (in the range of zero to one) is being fed into a multiplication block along with the flow from the silo. This creates a linear valve characteristic (ignoring the effect of backpressure on the flow rate through the valve) which is altering the flow into the evaporator feed tank dependant on the controller response. The PID block settings, if changed in Fix32 while the simulation is running, are used by the control block to alter the output of the model. In this fashion the control loops as they are on the plant are integrated into the model and connected to Fix32.

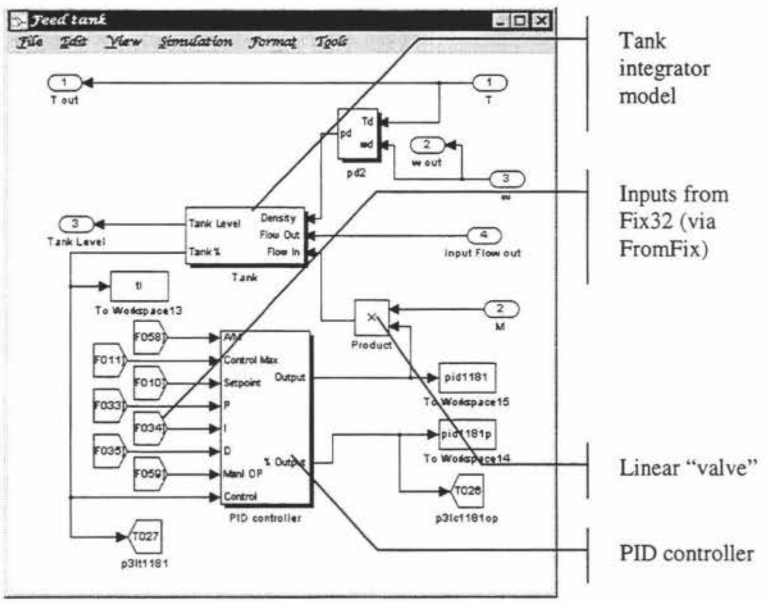


Figure 22 PID Controller integrated into the evaporator model (feed tank)

The control loop for the PID controller is created through the communication drivers, as shown in Figure 23. This structure was adopted because it allowed both the user to change the setpoints (quite important to the functioning of the simulator) and it allowed display of the current setpoint and control response.

This requires that present within Fix32 are the setpoint and PID values at initialisation, however. This is the responsibility of the InitFix driver. When the simulation is initialised, InitFix inserts the start-up values into Fix32, via EDA. When the simulation actually begins, and the controllers (via the FromFix driver) require values upon which to operate, they are present and ready.

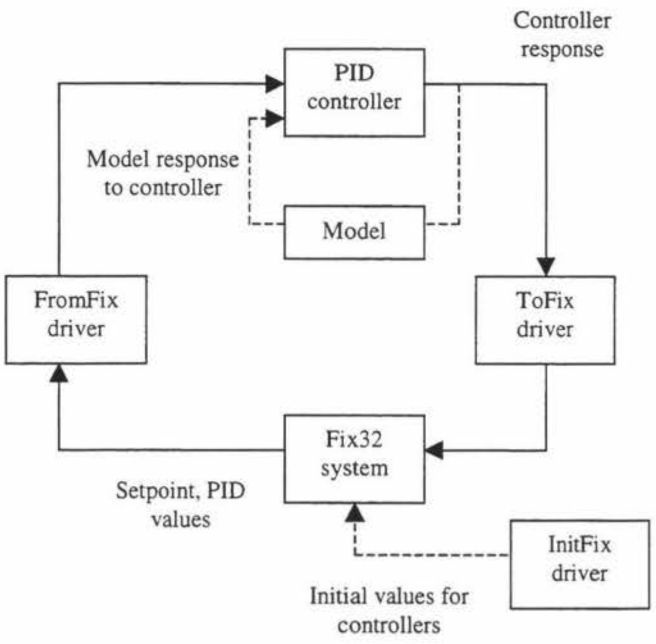


Figure 23 Control loop, through the communication drivers and PID controllers

The other modification made to the model was the insertion of the tanks and valves. As was the case with the PID controllers a generic tank block was created that could be configured to represent a tank of a given size based around an integrator and the flow into and the flow out of the tank. This subsystem block can then be dropped into the model and configured to match the physical tank it represents (Figure 22).

Valves were modelled as simple linear proportional valves and also ignored the effects of backpressure on the flow through the valve. This simplification was still accurate enough for the operators to use the simulator as they would the real plant.

One problem with the valves was that if the inputs to some of the equations in the model reduce to zero then the equations require a division by zero, causing an error in the simulation. This situation can easily develop if a valve is closed to zero. To avoid this, the valves in the model were implemented with a slight "leak". A small constant (0.001) was added to the percentage open for each of the valves; this ensured that for the purposes of the simulation the valves could not be fully closed and so no divide by zero errors could be generated.

2.3.2 Integrating the model and the software

Installation of the drivers into the model consists only of dragging each of the driver subsystems and the Driver Control Panel (DCP) into the top level of the model. Integration of the software into the operation of the model is slightly more involved but no more complex.

In order to connect ToFix into the operation of the model, and hence Fix32, it is necessary to identify points within the model that correspond to the location of sensors on the plant. For example, there is a temperature sensor just after the holding tubes in the preheat section of the evaporator which is used to measure the temperature of the milk prior to entering the bottom of the flash vessels. This temperature is predicted in the model by a set of differential equations. Fix32 normally expects this temperature to be in the control database in the Analogue Input (AI) block labelled P3TE1281. To connect them a Goto block (with a label pertaining to the ToFix driver) is inserted into the model taking the answer from the equations and making it available to the driver on a channel (a channel is defined as the path for a single value, either from or to Fix32). In the configuration file for the ToFix driver, this channel is then assigned to the P3TE1281 block.

The same process is undertaken for FromFix, except that a From block is inserted into the model as an input and the FromFix configuration file altered to contain the desired tag name for that channel in the driver.

2.3.3 Fix32 database modifications

The changes made to the Fix32 database are mainly the generic ones that have been discussed in the Fix32 Configuration section. These alterations are global changes made to all the blocks in the database simply to allow EDA calls to insert values from the model. In brief, they are the replacement of the input driver for each block with the internal SIM driver (and renumbering the memory addresses), the alteration of the block mode to MANL and the setting of the scan time for each block to a nominal value of one second.

A second set of changes was made to the database to create the model control screens and the values related to it. These are additions to the database in the form of Analogue Input (AI) blocks that are used to store the values for permutations to the inputs of the model. All of these blocks are labelled with the prefix "MC_" to indicate that they are related to model control, and can be altered from the SimEvap model control screen (see below).

A complete list of the changes to both the control screens and the database can be found in Appendix E : Fix32 Database and Control Screen Modifications, along with the purpose of each of the "MC_xxx" blocks.

2.3.4 Control screen modifications

Visually the control screens were modified as little as possible; the desire was for a simulator that looked as close to the "real thing" as possible. The visual modifications that were made were mainly the removal of indicators from the control screens that are not simulated in the model. An example of this is the condensate level sensor in the interior of the evaporator shell. The other visual modification made was the "greying out" of functions that are not supported in the main plant selection bar and in the pulldown menus from the selection bar. This restricts the operator to screens that relate to the simulator.

The largest modification made was the replacement of the node names. The control screens had been designed to operate from values stored in Fix32 databases such as POWDER, PSILO and P3A. All of these had to be replaced with the name of the simulator node; MODEL.

This also applied to the tag groups[11,22,23] used by the control screens. Tag groups are used to fill generic display structures where the only difference is the tag names. An example of this would be a generic temperature sensor display taking the value to display from a different tag in the database (alias "sensor on the plant"). These tags are stored in tag files with the TGE extension; the Fix32 control database for Powder 3 uses tag groups for the control loop and the sensors on the plant, and all of these groups had to be altered to refer to the MODEL node.

Dynamos[23] are another generic display tool; created as templates, they are used in the control screens to give a constant look and feel to the control system (examples of dynamos can be seen in Figure 24). Once the dynamo template is constructed and then inserted into the screen, it is configured with text strings to take values from the desired tags in the database. Unfortunately, the search and replace function used to turn make all of the node references refer to MODEL does not test text strings[24]. Consequently, each of the dynamos in the control screens had to be manually checked and altered.

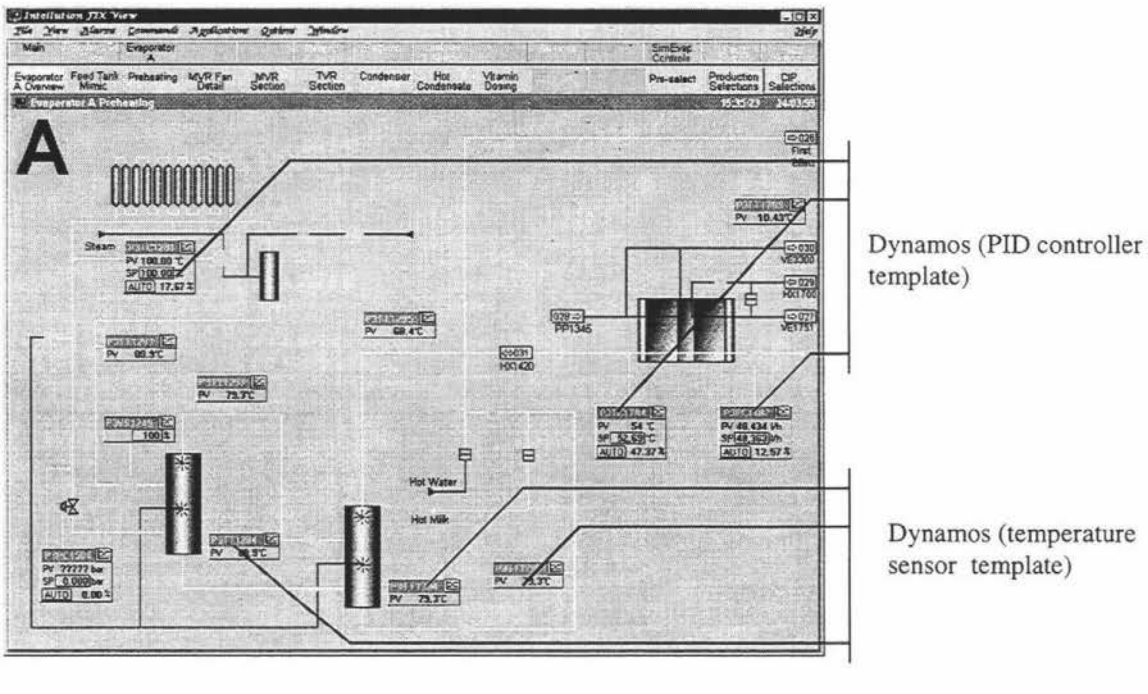

Dynamos (PID controller template)
Dynamos (temperature sensor template)
A
Steam
Hot Water
Hot Milk

Figure 24 Examples of Dynamos (Evaporator A Preheating screen)

A complete list of the changes to both the control screens and the database can be found in Appendix E : Fix32 Database and Control Screen Modifications.

2.3.5 SimEvap Control Screen

The model control screen is designed for the use of the operators to change the settings for the model – it does not alter the simulator software settings.

The control screen can be seen in Figure 25. It is reached via a button (coloured red to differentiate it from the normal controls) inserted into a blank section of the plant selection bar. The page itself shows a schematic of an evaporator representing the model. Variables that are available to be altered are located on the schematic in relation to what they manipulate. This approach was taken, as opposed to a list of variables and their values, in order to make the page as intuitive as possible to the operator.

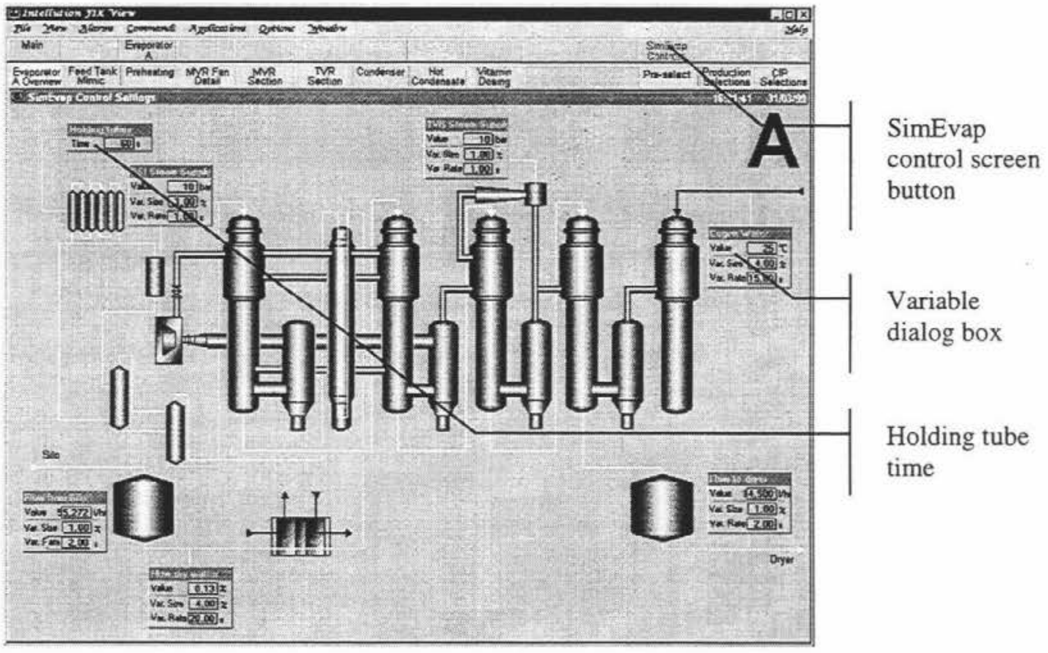


Figure 25 SimEvap control screen

The variable dialog boxes are Dynamos, and were adapted from the existing Dynamo used for PID controllers. Each variable Dynamo has three values – the nominal value of the variable, the size of fluctuations (%) in the variable and the speed of fluctuations (seconds). The variables were implemented in this fashion because in the real world there are also fluctuations on the variables concerned. The only exception to this is the holding tube time, which is a single constant value measured in seconds.

2.4 Testing

Testing of the software as a functional unit was carried out using simple testing of each communication point between the simulation software and the Fix32 control system, modified for use with the simulator.

Testing the validity of the model with respect to the response of the real plant was performed prior to the integration of the model and the simulation software. This testing was performed as part of a PhD in progress **references here.

Testing of the software was carried out informally with the help of operators randomly selected from those currently on shift at the Powder plant. This was done with a single operator at a time on a one-on-one basis. The simulator was started and the operator asked to "play with it". This approach seemed to relax the operator rather than have them feel like they were participating in a formal test. Their comments were noted and refinements made on that basis.

The comments made by the operators dealt with many aspects of the simulator, ranging from small cosmetic mistakes with valves that appeared open but were closed to steam pressures that were fluctuating far more than on the real plant. However, the overall impression was positive – the response and appearance of the simulator matched the "real thing" closely enough that some surprise was expressed.

Incidentally, from these discussions with the operators some flaws in the Fix32 control system were discovered that were passed on to the plant management. These flaws were then fixed in the control system of the real plant.

Part III: CONCLUSIONS AND RECOMMENDATIONS

1 SRTLink Related

The construction of SRTLink followed standard software engineering protocols. This process involved the laying down of a specification for the software, which included the number of tags to communicate and the desired frequency of communication. This was followed by a design, incorporating the specification. An implementation in Visual C led to the actual construction of the software, followed by testing to ensure the package was as bug free as possible and met the specifications. The end result was a software package that is able to communicate between a model of a process and the control system of that process in real-time.

Future developments of the software can lead in different directions, or simply increase the functionality of the software to allow broader applications. Some recommendations for further work include:

- Currently a single value taken from the model can only be output to a single tag in Fix32. In some situations it would be more efficient to be able to take a single value from the model and output it to multiple tags rather than using multiple input lines to achieve the same effect. Note that the ability to take a single value from Fix32 and output it on multiple lines it moot, as this can be achieved in Simulink with multiple From blocks on the channel.
- Within the current version there is no way to reset the simulation and then restart it. This can be achieved at a code level by simply breaking the simulation loop, rerunning the model shutdown and initialisation code and then re-entering the loop. A problem that needs to be surmounted with this advance is communication between the model level simulation and the code level operation, in order to trigger the reset. Currently no mechanism exists to do this.
- Although the compiled simulator works well as a console application, it would be much more user friendly to implement it as a Win32 application. This would allow access to all of the Windows API libraries and functions and also allow things like built-in editing of default settings for the application, rather than using a configuration file.

2 SimEvap Related

Once the development of SRTLink had been completed, SimEvap was created around SRTLink. This meant that a model of the process under question (a falling film milk evaporator) was necessary. Such a model was available, developed by James Winchester for the purposes of better understanding and hence optimising the evaporator. This model had also already been partially implemented in Simulink and was therefore perfect for use in order to construct a real-time simulator.

SimEvap then followed the same path as SRTLink; from specification through design, implementation and testing. The design of the simulator was based around the structure of SRTLink, as of needs it must be. The implementation consisted of adapting the model to use as a simulator and the embedding of the communication drivers in the model, to create the link to Fix32. Testing to ensure that the specifications were met was also carried out with the help of operators familiar with the actual control system on the falling film evaporator. The final result was an application that, when run with Fix32 created a simulator operating in real-time, to be used for operator training.

It should also be noted that recompilation of the model could be expected to happen at frequent intervals, whenever the evaporator is altered or changes are made to the plant control system. Recompilation involves making the alterations needed (such as the addition of more control and communication points, improvement of the model or updating the model process) and then running the new model through the RTW to generate the end application.

Future directions for SimEvap should be mostly model-based, unless improvements in the operation of SRTLink altered the basis for SimEvap. Examples of possible improvements are:

- General model improvements in order to match it more closely to the responses of the real plant. This could include such things as modelling blowback in the DSI and flooding of the flash vessels. Some milk products also involve bypassing one of the flash vessels; this cannot currently be modelled, although implementing it within Simulink should not be overly complicated.

- Addition of the recirculation loop, used for the switch from water to milk would be of benefit. Currently this can be simulated by altering the dry matter in the flow in the SimEvap Control page, but this method does not match the procedure used on the real plant (with respect to a new operator, training them in this fashion would almost be detrimental).
- Currently the pumps and valves are modelled very crudely, ignoring the effects of backpressure to flow. This can be remedied with a generic valve and pump block, correctly modelling the dynamics of each, which can then be inserted into the model at the relevant points.
- If an operator who is using the simulator wants to reset the "plant" at this stage, they need to close the SimEvap model application and then re-execute it. Ideally, there should be a "reset simulation" button on the SimEvap controls page to perform this function. This is currently not possible, because SRTLink cannot reset a model under its control. Improving SRTLink to allow this is, however one of the first improvements suggested for it. This development of SRTLink leads naturally to the addition of a reset for SimEvap, however.
- The ability to select a particular milk solution from the SimEvap Controls page would also be beneficial. Currently the selection of different milk solutions is done by altering the model to reflect the thermal properties of the different types of milk. This cannot be changed once the model has been compiled (apart from the percentage of milk solids), but is easily alterable within the model itself. This would necessitate a library of the properties for each of the milk types, possibly embedded in the Simulink model and selectable from the SimEvap Controls page.
- An alarm to indicate when the model is outside it's boundaries would also remove the guesswork currently necessary by the operator. However, conditions for activating the alarm could be difficult to implement. Checks on temperatures and flows to ensure they are within nominal ranges are easily performed, but monitoring the internal mechanics of the model for valid solutions poses more of a problem.

REFERENCES

1. Prais, M "Operator Training Simulator: Algorithms and Test Results", *IEEE Transactions on Power Systems*, August 1989, IEEE, New York

2. Rajagopal, S *et al* "Workstation based advanced operator training simulator for Consolidated Edison", *IEEE Transactions on Power Systems*, November 1994, IEEE, New York

3. Kambale, P "The dispatcher training simulator for Metropolitan Edison Company", *IEEE Transactions on Power Systems*, May 1996, IEEE, New York

4. Garlick, S "Application of same equation based model from steady state design to control system analysis and operator training", *Computers and Chemical Engineering*, Vol 18 1994, Pergamon Press, New York

5. Gross, F "Modelling, simulation and controllability analysis of an industrial heat-integrated distillation process", *Computers and Chemical Engineering* Vol 22 1998, Pergamon Press, New York

6. McCracken, G E "Gas controller trainer employs SCADA-linked simulation", *Pipe Line Industry*, July 1993

7. Gallant, John "Control system simulation: Simulation software gains sophistication", *EDN* Vol 37 No 9 1992

8. Wutkewicz, M "Plant wide control of an industrial process", *Chemical Engineering Research and Design*, Vol 76 No A2 1998, Pergamon Press, New York

9. Bakker, H H "Reducing development time for real-time simulations in a SCADA environment", *Institute of Professional Engineers of New Zealand Conference Proceedings - Simulation and Control*, Vol 2 1997, IPENZ, Wellington

10. Rao, R S *Integrating Real-Time Simulation Models into a SCADA environment* 1997 Masterate Thesis, Department of Production Technology Massey University, Palmerston North

11. "Introduction to Fix Systems", *System Setup*, version 6.0-10.95, Intellution, St. Louis Missouri

12. "EDA Reference", 1996, Windows help file [for Fix32 v6.0], Intellution, St. Louis Missouri

13. *Using Simulink 2.2*, 1998, PDF document supplied with Simulink 2.2, Mathworks, Natick Massachusetts

14. *RTW User's Guide 2.2*, 1998, PDF document supplied with Simulink 2.2, Mathworks, Natick Massachusetts

15. HTML block reference, released on CD with Simulink 2.2, Mathworks, Natick Massachusetts

16. "Database Builder", *System Development*, version 6.0-10.95, Intellution, St. Louis Missouri

17. *The Concise Oxford Dictionary* (8th Edition), 1990, Clarendon Press, Oxford

18. Fergusson, P H "Developments in the evaporation and drying of dairy products", *Journal of the Society of Dairy Technology*, Vol 42 No 4 1989, Plunkett Foundation Oxford

19. Winchester, J "Model based analysis of falling film evaporators", *Proceedings of the Institute of Professional Engineers of New Zealand Sustainable City Conference* 1997, IPENZ, Wellington

20. Quaak, P "Modelling dynamic behaviour of multiple-effect falling-film evaporators", *Computer Applications in Chemical Engineering* 1990, Elsevier, Amsterdam

21. Quaak, P et al "Comparison of process identification and physical modelling for falling-film evaporators", *Food Control* Vol 5 No 2 1994, Butterworth Scientific, London

22. "Tag Group Editor", *Display Development*, version 6.0-10.95, Intellution, St. Louis Missouri

23. "Draw", *Display Development*, version 6.0-10.95, Intellution, St. Louis Missouri

24. "View", *Display Development*, version 6.0-10.95, Intellution, St. Louis Missouri

ACRONYMS AND ABBREVIATIONS

AI	Analogue Input (block type in the Fix32 database)
DIT	Driver Image Table
DLL	Dynamically Linked Library
EDA	Easy Database Access
MVR	Mechanical Vapour Recompression
NTF	Node Tag Field
OS	Operating System
PDBSN	"PDB" Serial Number (refers to a particular Fix32 database)
PID	Proportional / Integral / Derivative
PLC	Programmable Logic Controller
RTW	Real-time Workshop
SAC	Scan, Alarm and Control
SCADA	Supervisory Control and Data Acquisition
SRT	"Sleeping" Real Time
TVR	Thermal Vapour Recompression
VSP	Variable Specification Parameter (used by Fix32 for defining NTF's)

APPENDIX A: CODE LISTING

The code lists here are not the entire source. Any sections of the code that are responsible for "housekeeping" (ie. were part of the original template), or that are not used but were left in the actual code for future expansion, have been removed. Fragments of code that are commented out for normal operation of the software (for example screen display code for debugging purposes) have been left in.

Any lines that overflowed onto the next line have been right-justified to allow for easier reading of the code, and to not disturb the flow and indenting.

ToFix.c Source Code

```
#define VERSION        1.32

// New defines and includes
#define  OS_WNT
#include "fixeda.h"
#include "fixtools.h"
#include "stdio.h"

#include <iostream.h>

// to read the config file name from the block parameters, use the line:
//   mxGetString(ssGetSFcnParam(S,0), CfgFile, CfgFileLength);
// where CfgFile is a string of lenfth CfgFileLength.

#define CONFIG_FILE_NAME    "tofix.cfg"
#define NUM_TAGS_IN         ssGetInputPortWidth(S,0)    // width of the mux feed into the driver
#define TF_SEP              "."                         // seperator for tag/field in the config file
#define DEF_NODE_NAME       "MODEL"                     // default node name

#define NUM_PARAMS          (0)      // number of parameters expected from the block
#define NTF_LIST_LENGTH     (300)    // max number of NTF's
#define NTF_STRING_SIZE     (40)     // max length of NTF strings
#define GH_IDX              (0)      // index in the PWork vector for passing GroupHandle
#define NTF_IDX             (1)      // index in the PWork vector for passing NTFHandle
#define NN_IDX              (2)      // index in the PWork vector for passing NodeName
#define TL_IDX              (3)      // index in the PWork vector for passing TagList
#define FL_IDX              (4)      // index in the PWork vector for passing FieldList
#define NTFC_IDX            (5)      // index in the PWork vector for passing NTFCount
#define PNTFL_IDX           (6)      // index in the PWork vector for passing PackedNTFList
#define VNTFL_IDX           (7)      // index in the PWork vector for passing ValidNTFList
#define VNTFC_IDX           (8)      // index in the PWork vector for passing VNTFCount

#define PWORK_SIZE          (9)      // size of PWork Vector (1+largest index value above)
#define RWORK_SIZE          (0)      // size of RWork Vector
#define IWORK_SIZE          (0)      // size of IWork Vector

#define DATA_PORT           (0)      // defines which input port is for data

#define CONTROL_PORT        (1)      // defines which input port is the control port
#define CP_WIDTH            (1)      // width of control port
#define CP_ENABLE           (0)      // index in ControlPort for enable toggle
```

```
// standard defines and includes
#define S_FUNCTION_LEVEL    2
#define S_FUNCTION_NAME tofix

#include "simstruc.h"

/*===================*
 * S-function methods *
 *===================*/

/* Function: mdlInitializeSizes ===========================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
     ssSetNumSFcnParams(S, NUM_PARAMS);  /* Number of expected parameters */
     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
     }

    // not used at all, as there are no "states" as such
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 2)) return;
    ssSetInputPortWidth(S, DATA_PORT, DYNAMICALLY_SIZED);
    ssSetInputPortWidth(S, CONTROL_PORT, CP_WIDTH);

     ssSetInputPortDirectFeedThrough(S, DATA_PORT, 1);
     ssSetInputPortDirectFeedThrough(S, CONTROL_PORT, 1);

    if (!ssSetNumOutputPorts(S, 0)) return;
    // ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, DYNAMICALLY_SIZED);
    ssSetNumIWork(S, DYNAMICALLY_SIZED);
    ssSetNumPWork(S, DYNAMICALLY_SIZED);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetOptions(S, 0);
}

/* Function: mdlInitializeSampleTimes =====================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MDL_SET_WORK_WIDTHS
#if defined(MDL_SET_WORK_WIDTHS)
  /* Function: mdlSetWorkWidths ==========================================
   * Abstract:
   *    Used to define the size of the input/output ports, and the work vectors
   */
    static void mdlSetWorkWidths(SimStruct *S)
    {
        ssSetNumRWork(S,RWORK_SIZE);
        ssSetNumIWork(S,IWORK_SIZE);
        ssSetNumPWork(S,PWORK_SIZE);
    }
#endif /* MDL_SET_WORK_WIDTHS */

#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart ===================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
static void mdlStart(SimStruct *S)
{
    // variables that are common between mdlStart and mdlOutputs (ie. passed with PWork)
    void                    **PWork=ssGetPWork(S);
    static GNUM             GroupHandle;
    static THANDLE          NTFHandle[NTF_LIST_LENGTH];
    static char             NodeName[NTF_STRING_SIZE],
                            TagList[NTF_LIST_LENGTH][NTF_STRING_SIZE],
                            FieldList[NTF_LIST_LENGTH][NTF_STRING_SIZE];
    static unsigned int     PackedNTFList[NTF_LIST_LENGTH],
                            ValidNTFList[NTF_LIST_LENGTH],
                            NTFCount=0,VNTFCount=0;

    // other variables
    static long             key = KEY_ADI_TYPE | KEY_ALL_AREAS;
    static unsigned int     i,NTFIndex=0,ListToggle=0;
    FILE                    *CfgStream;
    long int                LineCount=0;
    char                    StringIn[100],*TokenStr,*ParamStr,*SettingStr,*TempT,*TempF,
                            msg[256];
    INT16                   err;

    // InputRealPtrsType         ControlPort = ssGetInputPortRealSignalPtrs(S,1);

    /* Currently Simulink doesn't fill the port vectors until after the
       initalisation of the s-function is finished (ie. mdlStart has completed
       execution).  However, the enable/disable code is left here commented out,
       and left in the mdlOutputs function to at least disable the s-function
       during simulation.  The code is also in mdlTerminate, also commented out.
    */

    // *********************************************
    // Enable On/Off check
    //if (!(*ControlPort[CP_ENABLE])) return;
    // *********************************************

    printf("\nToFix EDA I/O Driver for Simulink/RTW: Version %.2f\n",VERSION);

    if (NUM_TAGS_IN>NTF_LIST_LENGTH) {
        ssSetErrorStatus(S,"ToFix Fatal error: Internal tag limit overflow\n");
        ssSetStopRequested(S,1);
        return;
    }

    GroupHandle=eda_define_group(1,0);

    if(!GroupHandle) {
        ssSetErrorStatus(S,"ToFix Fatal error: Could not define group handle\n");
        ssSetStopRequested(S,3);
        return;
    }

    eda_set_key(GroupHandle,key);     // not strictly necessary, but included for pre-version 3 Fix

    strncpy(NodeName,DEF_NODE_NAME,NTF_STRING_SIZE);     // set NodeName to default in case not set in config

    if ((CfgStream=fopen(CONFIG_FILE_NAME,"r"))==NULL) {
        printf("ToFix error: Could not find config file %s - ToFix disabled\n",CONFIG_FILE_NAME);
    } else {
        while (!feof(CfgStream)) {
            if (fgets(StringIn,100, CfgStream)==NULL) {
                LineCount++;
                if (!feof(CfgStream)) {
                    printf("ToFix: Config file error line %i - Undetermined\n",LineCount);
                }
            } else {  // parse the token
                LineCount++;
                if (strncmp(StringIn,";",1)!=0) { // if the line is not a comment then parse

                    TokenStr=strtok(StringIn,";"); // cut out comments anywhere else in line
                    TokenStr=_strlwr(TokenStr);
                    if (TokenStr!=NULL) ParamStr=strtok(TokenStr," =\t\n;"); // find the parameter
                    if (ParamStr!=NULL)     SettingStr=strtok(NULL," =\t\n;"); // find the setting

                    // printf("%s | %s\n",ParamStr,SettingStr);
                    if ((ParamStr!=NULL) && (SettingStr!=NULL)) {

                        if (!strcmp(ParamStr,"nodename")) {
                        // no checking is done for the validity of the node name, truncated to
                            strncpy(_strupr(NodeName),SettingStr,NTF_STRING_SIZE);
                        }

                        NTFIndex=atoi(ParamStr);
                        if ((NTFIndex>0) && (NTFIndex<=NTF_LIST_LENGTH)) {

                        TempT=strtok(SettingStr,TF_SEP);
                        TempF=strtok(NULL,TF_SEP);

                        if ((TempT!=NULL) && (TempF!=NULL)) {
                            NTFIndex--;         // so that it's looking at the right place in the arrays
                            strncpy(TagList[NTFIndex],_strupr(TempT),NTF_STRING_SIZE);
                            strncpy(FieldList[NTFIndex],_strupr(TempF),NTF_STRING_SIZE);
                            PackedNTFList[NTFCount]=NTFIndex;
                            NTFCount++;
                            } else {
                                printf("ToFix: Illegal tagfield on line %i\n",LineCount);
                            }
                        } else {
                            // if tagindex is too large, error
                            if (NTFIndex>NTF_LIST_LENGTH) {
                                printf("ToFix: Tag Index too large line %i - tag will not be used\n",
                                                                                          LineCount);
                            }
                        }
                    } // end of null check parse
                } // end of token parsing
            } // end of check for fgets error
        } // end of file
        fclose(CfgStream);
    }

// debug display
/*
    printf("\ni\tPackedNTFList\tTag\tField\n");
    for (i=0;i<NTFCount;i++) {

            printf("%i\t%i\t\t%s\t%s\n",i,PackedNTFList[i],TagList[PackedNTFList[i]],FieldList[PackedNTFLi
        st[i]]);
    }

*/

    // ************************************************************
    // prep GroupHandle to be used in mdlOutputs
    for (i=0;i<NTFCount;i++) {
        NTFHandle[PackedNTFList[i]]=eda_define_ntf(GroupHandle,NodeName,TagList[PackedNTFList[i]],
                                                                 FieldList[PackedNTFList[i]],0);
        if (NTFHandle[i]==BAD_THANDLE) {
            printf("ToFix: Bad handle %s.%s\n",TagList[i],FieldList[i]);
        }
    }

    eda_lookup(GroupHandle);
    eda_wait(GroupHandle);
    for (i=0;i<NTFCount;i++) {
        err=eda_get_error(GroupHandle,NTFHandle[PackedNTFList[i]]);
        if (err!=FE_OK) {
            NlsGetText(err,msg,256);
            printf("ToFix: Tag %s.%s
        (%i)error\n%s\n",TagList[PackedNTFList[i]],FieldList[PackedNTFList[i]],
                                                                      PackedNTFList[i]+1,msg);
        } else {
            ValidNTFList[VNTFCount]=PackedNTFList[i];
            VNTFCount++;
        }
    }
    printf("Tags read: %i\t\tTags confirmed: %i\n",NTFCount,VNTFCount);

    PWork[GH_IDX]=GroupHandle;
    PWork[NTF_IDX]=NTFHandle;
    PWork[NN_IDX]=NodeName;
    PWork[TL_IDX]=TagList;
    PWork[FL_IDX]=FieldList;
    PWork[NTFC_IDX]=&NTFCount;
    PWork[PNTFL_IDX]=PackedNTFList;
    PWork[VNTFL_IDX]=ValidNTFList;
    PWork[VNTFC_IDX]=&VNTFCount;
}
#endif /*  MDL_START */

/* Function: mdlOutputs =======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid) {
    // variables that are common between mdlStart and mdlOutputs (ie. passed with PWork)
    static GNUM             GroupHandle;
    static THANDLE          *NTFHandle;
    static char             *NodeName,
                            (*TagList)[NTF_STRING_SIZE],
                            (*FieldList)[NTF_STRING_SIZE];
    static unsigned int     *PackedNTFList,*ValidNTFList,*NTFCount,*VNTFCount;

    // other variables
    INT16                   err;
    InputRealPtrsType       uPtrs = ssGetInputPortRealSignalPtrs(S,0),
                            ControlPort = ssGetInputPortRealSignalPtrs(S,1);
    static char             msg[256];
    float                   Value;
    static unsigned int     i,NTFIndex;

    // *********************************************
    // Enable On/Off check
    if (!(*ControlPort[CP_ENABLE])) return;
    // *********************************************

    // printf("Time: %f\n",ssGetT(S));    // check for triggering interval of driver

    // code to read pointers out of PWork, and cast them back to what they should be
    GroupHandle   = ssGetPWorkValue(S,GH_IDX);
    NTFHandle = ssGetPWorkValue(S,NTF_IDX);
    NodeName = ssGetPWorkValue(S,NN_IDX);
    TagList = (char (*)[NTF_STRING_SIZE])ssGetPWorkValue(S,TL_IDX);
    FieldList = (char (*)[NTF_STRING_SIZE])ssGetPWorkValue(S,FL_IDX);
    NTFCount = ssGetPWorkValue(S,NTFC_IDX);
    PackedNTFList = ssGetPWorkValue(S,PNTFL_IDX);
    ValidNTFList = ssGetPWorkValue(S,VNTFL_IDX);
    VNTFCount = ssGetPWorkValue(S,VNTFC_IDX);

    for (i=0;i<*VNTFCount;i++) {
        NTFIndex=ValidNTFList[i];
        Value=(float)*uPtrs[NTFIndex];

        err=eda_set_float(GroupHandle,NTFHandle[NTFIndex],&Value);
    }

    eda_write(GroupHandle);
    eda_wait(GroupHandle);

    for (i=0;i<*VNTFCount;i++) {
        NTFIndex=ValidNTFList[i];
        err=eda_get_error(GroupHandle,NTFHandle[NTFIndex]);
        if (err!=FE_OK) {
            NlsGetText(err,msg,256);
            printf("ToFix: Tag %s.%s (%i) error -
        %s\n",TagList[NTFIndex],FieldList[NTFIndex],NTFIndex+1,msg);
        }
    }
}
```

```
/* Function: mdlTerminate =====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlInitializeConditions, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
    static GNUM        GroupHandle;
    GroupHandle = ssGetPWorkValue(S,GH_IDX);

    // ********************************************
    // Enable On/Off check
    //if (!(*ControlPort[CP_ENABLE])) return;
    // ********************************************

    if (GroupHandle!=NULL) {
        eda_delete_group(GroupHandle);
        printf("ToFix: EDA group released\n");
    }

}
```

FromFix.c Source Code

```
#define VERSION        1.27

// New defines and includes
#define  OS_WNT
#include "fixeda.h"
#include "fixtools.h"
#include "stdio.h"

#include <iostream.h>

// to read the config file name from the block parameters, use the line:
//   mxGetString(ssGetSFcnParam(S,0), CfgFile, CfgFileLength);
// where CfgFile is a string of lenfth CfgFileLength.

#define CONFIG_FILE_NAME     "fromfix.cfg"
#define NUM_TAGS_OUT         ssGetOutputPortWidth(S,0)   // width of the mux feed into the driver
#define TF_SEP               "."                         // seperator for tag/field in the config file
#define DEF_NODE_NAME        "MODEL"                     // default node name

#define NUM_PARAMS           (0)     // number of parameters expected from the block
#define NTF_LIST_LENGTH      (300)   // max number of NTF's
#define NTF_STRING_SIZE      (40)    // max length of NTF strings
#define GH_IDX               (0)     // index in the PWork vector for passing GroupHandle
#define NTF_IDX              (1)     // index in the PWork vector for passing NTFHandle
#define NN_IDX               (2)     // index in the PWork vector for passing NodeName
#define TL_IDX               (3)     // index in the PWork vector for passing TagList
#define FL_IDX               (4)     // index in the PWork vector for passing FieldList
#define NTFC_IDX             (5)     // index in the PWork vector for passing NTFCount
#define PNTFL_IDX            (6)     // index in the PWork vector for passing PackedNTFList
#define VNTFL_IDX            (7)     // index in the PWork vector for passing ValidNTFList
#define VNTFC_IDX            (8)     // index in the PWork vector for passing VNTFCount

#define PWORK_SIZE           (9)
#define RWORK_SIZE           (0)
#define IWORK_SIZE           (0)

#define CONTROL_PORT         (0)     // defines which input port is the control port
#define CP_SIZE              (1)     // width of control port
#define CP_ENABLE            (0)     // index in ControlPort for enable toggle

#define BUFFER_LENGTH        (40)    // length of ASCII I/O buffer to read values

// standard defines and includes
#define S_FUNCTION_LEVEL     2
#define S_FUNCTION_NAME      fromfix

#include "simstruc.h"
```

```
/*====================*
 * S-function methods *
 *====================*/

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
     ssSetNumSFcnParams(S, NUM_PARAMS);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }

     // not used at all, as there are no "states" as such
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, CONTROL_PORT, CP_SIZE);
    ssSetInputPortDirectFeedThrough(S, CONTROL_PORT, 1);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, DYNAMICALLY_SIZED);
    ssSetNumIWork(S, DYNAMICALLY_SIZED);
    ssSetNumPWork(S, DYNAMICALLY_SIZED);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetOptions(S, 0);
}

/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, ssGetStepSize(S));
    ssSetOffsetTime(S, 0, 0.0);

}

#define MDL_SET_WORK_WIDTHS
#if defined(MDL_SET_WORK_WIDTHS)
  /* Function: mdlSetWorkWidths ===============================================
   * Abstract:
   *    Used to define the size of the input/output ports, and the work vectors
   */
    static void mdlSetWorkWidths(SimStruct *S)
    {
         ssSetNumRWork(S,RWORK_SIZE);
         ssSetNumIWork(S,IWORK_SIZE);
         ssSetNumPWork(S,PWORK_SIZE);
    }
#endif /* MDL_SET_WORK_WIDTHS */

#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart =======================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
static void mdlStart(SimStruct *S)
{
     // variables that are common between mdlStart and mdlOutputs (ie. passed with PWork)
```

```
void                    **PWork=ssGetPWork(S);
static GNUM             GroupHandle;
static THANDLE          NTFHandle[NTF_LIST_LENGTH];
static char             NodeName[NTF_STRING_SIZE],
                        TagList[NTF_LIST_LENGTH][NTF_STRING_SIZE],
                        FieldList[NTF_LIST_LENGTH][NTF_STRING_SIZE];
static unsigned int     PackedNTFList[NTF_LIST_LENGTH],
                        ValidNTFList[NTF_LIST_LENGTH],
                        NTFCount=0,VNTFCount=0;

// other variables
static long             key = KEY_ADI_TYPE | KEY_ALL_AREAS;
static unsigned int     i,NTFIndex=0;
FILE                    *CfgStream;
long int                LineCount=0;
char                    StringIn[100],*TokenStr,*ParamStr,*SettingStr,*TempT,*TempF,
                        msg[256];
INT16                   err;
// InputRealPtrsType    ControlPort = ssGetInputPortRealSignalPtrs(S,1);

/* Currently Simulink doesn't fill the port vectors until after the
   initalisation of the s-function is finished (ie. mdlStart has completed
   execution).  However, the enable/disable code is left here commented out,
   and left in the mdlOutputs function to at least disable the s-function
   during simulation.  The code is also in mdlTerminate, also commented out.
*/

// **********************************************
// Enable On/Off check
//if (!(*ControlPort[CP_ENABLE])) return;
// **********************************************

printf("\nFromFix EDA I/O Driver for Simulink/RTW: Version %.2f\n",VERSION);

if (NUM_TAGS_OUT>NTF_LIST_LENGTH) {
    ssSetErrorStatus(S,"FromFix Fatal error: Internal tag limit overflow\n");
    ssSetStopRequested(S,1);
    return;
}

GroupHandle=eda_define_group(1,0);

if(!GroupHandle) {
    ssSetErrorStatus(S,"FromFix Fatal error: Could not define group handle\n");
    ssSetStopRequested(S,3);
    return;
}

eda_set_key(GroupHandle,key);    // not strictly necessary, but included for pre-version 3 Fix

strncpy(NodeName,DEF_NODE_NAME,NTF_STRING_SIZE);    // set NodeName to default in case not set in file

if ((CfgStream=fopen(CONFIG_FILE_NAME,"r"))==NULL) {
    printf("FromFix error: Could not find config file %s - FromFix disabled\n",CONFIG_FILE_NAME);
} else {
    while (!feof(CfgStream)) {
        if (fgets(StringIn,100, CfgStream)==NULL) {
            LineCount++;
            if (!feof(CfgStream)) {
                printf("FromFix: Config file error line %i - Undetermined\n",LineCount);
            }
        } else {  // parse the token
            LineCount++;
            if (strncmp(StringIn,";",1)!=0) { // if the line is not a comment then parse

                TokenStr=strtok(StringIn,";"); // cut out comments anywhere else in line
                TokenStr=_strlwr(TokenStr);

                if (TokenStr!=NULL) ParamStr=strtok(TokenStr," =\t\n;"); // find the parameter
                if (ParamStr!=NULL)     SettingStr=strtok(NULL," =\t\n;"); // find the setting

                if ((ParamStr!=NULL) && (SettingStr!=NULL)) {
                    // log file name
                    if (!strcmp(ParamStr,"nodename")) {
                        // no checking is done for the validity of the node name, truncated to
                        strncpy(_strupr(NodeName),SettingStr,NTF_STRING_SIZE);
                        }

                        NTFIndex=atoi(ParamStr);

                        if ((NTFIndex>0) && (NTFIndex<=NTF_LIST_LENGTH)) {

                            TempT=strtok(SettingStr,TF_SEP);
                            TempF=strtok(NULL,TF_SEP);

                            if ((TempT!=NULL) && (TempF!=NULL)) {
                                NTFIndex--;          // so that it's looking at the right place in the
        arrays
                                strncpy(TagList[NTFIndex],_strupr(TempT),NTF_STRING_SIZE);
                                strncpy(FieldList[NTFIndex],_strupr(TempF),NTF_STRING_SIZE);
                                PackedNTFList[NTFCount]=NTFIndex;
                                NTFCount++;
                            } else {
                                printf("FromFix: Illegal tagfield on line %i\n",LineCount);
                            }
                        } else {
                            if (NTFIndex>NTF_LIST_LENGTH) {
                                printf("FromFix: Tag Index too large line %i - tag will not be
                                                                        used\n",LineCount);
                            }
                        }

                    } // end of null check parse
                } // end of token parsing
            } // end of check for fgets error
        } // end of file
        fclose(CfgStream);
    }

    for (i=0;i<NTFCount;i++) {
        NTFHandle[PackedNTFList[i]]=eda_define_ntf(GroupHandle,NodeName,TagList[PackedNTFList[i]],
                                                        FieldList[PackedNTFList[i]],0);
        if (NTFHandle[i]==BAD_THANDLE) {
            printf("FromFix: Bad handle %s.%s\n",TagList[i],FieldList[i]);
        }
    }

    eda_lookup(GroupHandle);
    eda_wait(GroupHandle);
    for (i=0;i<NTFCount;i++) {
        err=eda_get_error(GroupHandle,NTFHandle[PackedNTFList[i]]);
        if (err!=FE_OK) {
            NlsGetText(err,msg,256);
            printf("FromFix: Tag %s.%s (%i) error\n%s\n",TagList[PackedNTFList[i]],
                                    FieldList[PackedNTFList[i]],PackedNTFList[i]+1,msg);
        } else {
            ValidNTFList[VNTFCount]=PackedNTFList[i];
            VNTFCount++;
        }
    }

    printf("Tags read: %i\tTags confirmed: %i\n",NTFCount,VNTFCount);

    PWork[GH_IDX]=GroupHandle;
    PWork[NTF_IDX]=NTFHandle;
    PWork[NN_IDX]=NodeName;
    PWork[TL_IDX]=TagList;
    PWork[FL_IDX]=FieldList;
    PWork[NTFC_IDX]=&NTFCount;
    PWork[PNTFL_IDX]=PackedNTFList;
    PWork[VNTFL_IDX]=ValidNTFList;
    PWork[VNTFC_IDX]=&VNTFCount;
}
#endif /*  MDL_START */

/* Function: mdlOutputs =======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    // variables that are common between mdlStart and mdlOutputs (ie. passed with PWork)
    static GNUM                 GroupHandle;
    static THANDLE              *NTFHandle;
    static char                 *NodeName,
                                (*TagList) [NTF_STRING_SIZE],
                                (*FieldList) [NTF_STRING_SIZE];
    static unsigned int         *PackedNTFList, *ValidNTFList, *NTFCount, *VNTFCount;

    // other variables
    INT16                       err;
    real_T                      *y = ssGetOutputPortRealSignal(S,0);
    static char                 msg[256];
    char                        Buffer[BUFFER_LENGTH],FiltBuff[BUFFER_LENGTH],*Token;
    float                       Value;
    static unsigned int         i,NTFIndex;
    InputRealPtrsType           ControlPort = ssGetInputPortRealSignalPtrs(S,CONTROL_PORT);

    // ********************************************
    // Enable On/Off check
    if (!(*ControlPort[CP_ENABLE])) return;
    // ********************************************

    // code to read pointers out of PWork, and cast them back to what they should be
    GroupHandle   = ssGetPWorkValue(S,GH_IDX);
    NTFHandle = ssGetPWorkValue(S,NTF_IDX);
    NodeName = ssGetPWorkValue(S,NN_IDX);
    TagList = (char (*) [NTF_STRING_SIZE])ssGetPWorkValue(S,TL_IDX);
    FieldList = (char (*) [NTF_STRING_SIZE])ssGetPWorkValue(S,FL_IDX);
    NTFCount = ssGetPWorkValue(S,NTFC_IDX);
    PackedNTFList = ssGetPWorkValue(S,PNTFL_IDX);
    ValidNTFList = ssGetPWorkValue(S,VNTFL_IDX);
    VNTFCount = ssGetPWorkValue(S,VNTFC_IDX);

    eda_read(GroupHandle);
    eda_wait(GroupHandle);

    for (i=0;i<*VNTFCount;i++) {
        NTFIndex=ValidNTFList[i];

        if (strncmp(FieldList[NTFIndex],"A_",2)==0) {
            err=eda_get_ascii(GroupHandle,NTFHandle[NTFIndex],Buffer,BUFFER_LENGTH);

            // code to remove commas from a number (!)
            strcpy(FiltBuff,"");
            Token=strtok(Buffer,",");
            while (Token!=NULL) {
                strcat(FiltBuff,Token);
                Token=strtok(NULL,",");
            }

            Value=(float)atof(FiltBuff);
        } else {
                err=eda_get_float(GroupHandle,NTFHandle[NTFIndex],&Value);
        }

        if (err==FE_OK) {
            y[NTFIndex]=Value;
        }
    }
}

/* Function: mdlTerminate =============================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlInitializeConditions, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
    static GNUM        GroupHandle;
    GroupHandle = ssGetPWorkValue(S,GH_IDX);

    // ********************************************
    // Enable On/Off check
    //if (!(*ControlPort[CP_ENABLE])) return;
    // ********************************************
    if (GroupHandle!=NULL) {
        eda_delete_group(GroupHandle);
        printf("FromFix: EDA group released\n");
    }
}
```

InitFix.c Source Code

```
#define VERSION        1.01

// New defines and includes
#define  OS_WNT
#include "fixeda.h"
#include "fixtools.h"
#include "stdio.h"

#include <iostream.h>

// to read the config file name from the block parameters, use the line:
//   mxGetString(ssGetSFcnParam(S,0), CfgFile, CfgFileLength);
// where CfgFile is a string of lenfth CfgFileLength.

#define CONFIG_FILE_NAME     "initfix.cfg"
#define NUM_TAGS_IN          ssGetInputPortWidth(S,0)    // width of the mux feed into the driver
#define TF_SEP               "."                         // seperator for tag/field in the config file
#define DEF_NODE_NAME        "MODEL"                     // default node name

#define NUM_PARAMS           (0)      // number of parameters expected from the block
#define NTF_LIST_LENGTH      (300)    // max number of NTF's
#define NTF_STRING_SIZE      (40)     // max length of NTF strings

#define PWORK_SIZE           (0)      // size of PWork Vector (1+largest index value above)
#define RWORK_SIZE           (0)      // size of RWork Vector
#define IWORK_SIZE           (0)      // size of IWork Vector

#define CONTROL_PORT         (0)      // defines which input port is the control port
#define CP_WIDTH             (1)      // width of control port
#define CP_ENABLE            (0)      // index in ControlPort for enable toggle

#define ONCE_PORT            (0)      // number of output port used for writeonce list

// standard defines and includes
#define S_FUNCTION_LEVEL     2
#define S_FUNCTION_NAME      initfix

#include "simstruc.h"

/*====================*
 * S-function methods *
 *====================*/

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
     ssSetNumSFcnParams(S, NUM_PARAMS);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }

     // not used at all, as there are no "states" as such
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, CONTROL_PORT, CP_WIDTH);

     ssSetInputPortDirectFeedThrough(S, CONTROL_PORT, 1);
    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, ONCE_PORT, DYNAMICALLY_SIZED);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, DYNAMICALLY_SIZED);
    ssSetNumIWork(S, DYNAMICALLY_SIZED);
    ssSetNumPWork(S, DYNAMICALLY_SIZED);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    ssSetOptions(S, 0);
}



/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MDL_SET_WORK_WIDTHS
#if defined(MDL_SET_WORK_WIDTHS)
  /* Function: mdlSetWorkWidths ===============================================
   * Abstract:
   *    Used to define the size of the input/output ports, and the work vectors
   */
    static void mdlSetWorkWidths(SimStruct *S)
    {
         ssSetNumRWork(S,RWORK_SIZE);
         ssSetNumIWork(S,IWORK_SIZE);
         ssSetNumPWork(S,PWORK_SIZE);
    }
#endif /* MDL_SET_WORK_WIDTHS */


#undef MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
#if defined(MDL_INITIALIZE_CONDITIONS)
  /* Function: mdlInitializeConditions ========================================
   * Abstract:
   *    In this function, you should initialize the continuous and discrete
   *    states for your S-function block.  The initial states are placed
   *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
   *    You can also perform any other initialization activities that your
   *    S-function may require. Note, this routine will be called at the
   *    start of simulation and if it is present in an enabled subsystem
   *    configured to reset states, it will be call when the enabled subsystem
   *    restarts execution to reset the states.
   */
    static void mdlInitializeConditions(SimStruct *S)
    {
    }
#endif /* MDL_INITIALIZE_CONDITIONS */



#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart =======================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
static void mdlStart(SimStruct *S)
{
    // other variables
    static long            key = KEY_ADI_TYPE | KEY_ALL_AREAS;
    static unsigned int    i,NTFIndex=0,ListToggle=0;
```

```
FILE                    *CfgStream;
long int                LineCount=0;
char                    StringIn[100],*TokenStr,*ParamStr,*SettingStr,*TempT,*TempF,
                        msg[256],NodeName[NTF_STRING_SIZE];
INT16                   err;

// variables used in write once list
GNUM                    OnceHandle;
THANDLE                 OnceNTF[NTF_LIST_LENGTH];
float                   OnceValue[NTF_LIST_LENGTH],TempValue;
char                    OnceTL[NTF_LIST_LENGTH][NTF_STRING_SIZE],
                        OnceFL[NTF_LIST_LENGTH][NTF_STRING_SIZE],
                        *TempV;
unsigned int            OnceCount=0,OnceDone=0,PackedOnceList[NTF_LIST_LENGTH];
real_T                  *y = ssGetOutputPortRealSignal(S,ONCE_PORT);

// InputRealPtrsType    ControlPort = ssGetInputPortRealSignalPtrs(S,1);

/* Currently Simulink doesn't fill the port vectors until after the
   initalisation of the s-function is finished (ie. mdlStart has completed
   execution).  However, the enable/disable code is left here commented out,
   and left in the mdlOutputs function to at least disable the s-function
   during simulation.  The code is also in mdlTerminate, also commented out.
*/

// ********************************************
// Enable On/Off check
//if (!(*ControlPort[CP_ENABLE])) return;
// ********************************************

printf("\nInitFix EDA I/O Driver for Simulink/RTW: Version %.2f\n",VERSION);

if (NUM_TAGS_IN>NTF_LIST_LENGTH) {
    ssSetErrorStatus(S,"InitFix Fatal error: Internal tag limit overflow\n");
    ssSetStopRequested(S,1);
    return;
}

strncpy(NodeName,DEF_NODE_NAME,NTF_STRING_SIZE);    // set NodeName to default in case not set in file

if ((CfgStream=fopen(CONFIG_FILE_NAME,"r"))==NULL) {
    printf("InitFix error: Could not find config file %s - InitFix disabled\n",CONFIG_FILE_NAME);
} else {
    while (!feof(CfgStream)) {
        if (fgets(StringIn,100, CfgStream)==NULL) {
            LineCount++;
            if (!feof(CfgStream)) {
                printf("InitFix: Config file error line %i - Undetermined\n",LineCount);
            }
        } else {  // parse the token
            LineCount++;
            if (strncmp(StringIn,";",1)!=0) { // if the line is not a comment then parse

                TokenStr=strtok(StringIn,";"); // cut out comments anywhere else in line
                TokenStr=_strlwr(TokenStr);

                if (TokenStr!=NULL) ParamStr=strtok(TokenStr," =\t\n;"); // find the parameter
                if (ParamStr!=NULL)     SettingStr=strtok(NULL," =\t\n;"); // find the setting

                // printf("%s | %s\n",ParamStr,SettingStr);

                if ((ParamStr!=NULL) && (SettingStr!=NULL)) {

                    if (!strcmp(ParamStr,"nodename")) {
                    // no checking is done for the validity of the node name, truncated to
                        strncpy(_strupr(NodeName),SettingStr,NTF_STRING_SIZE);
                    }

                    NTFIndex=atoi(ParamStr);

                    if ((NTFIndex>0) && (NTFIndex<=NTF_LIST_LENGTH)) {

                        TempT=strtok(SettingStr,TF_SEP);
                        TempF=strtok(NULL,", ");
                        TempV=strtok(NULL,", ");
```

```
                            // printf("Setting:%s |TempT:%s |TempF: %s |TempV: %s\n",
                                                            SettingStr,TempT,TempF,TempV);

                            if ((TempT!=NULL) && (TempF!=NULL)) {
                                NTFIndex--;        // so that it's looking at the right place in the
        arrays
                                strncpy(OnceTL[NTFIndex],_strupr(TempT),NTF_STRING_SIZE);
                                strncpy(OnceFL[NTFIndex],_strupr(TempF),NTF_STRING_SIZE);
                                PackedOnceList[OnceCount]=NTFIndex;
                                OnceValue[NTFIndex]=(float)atof(TempV);
                                OnceCount++;

                            } else {
                                printf("InitFix: Illegal default tagfield on line %i\n",LineCount);
                            }
                        } else {
                            if (NTFIndex>NTF_LIST_LENGTH) {
                                printf("InitFix: Default tag index too large line %i - tag will not be
                                                                            used\n",LineCount);
                            }
                        } // end of list length check
                    } // end of null check parse
                } // end of token parsing
            } // end of check for fgets error
        } // end of file
        fclose(CfgStream);
    }

    // ************************************************************
    // do the writeonce list stuff
    OnceHandle=eda_define_group(1,0);
    if (OnceCount>0) {
        for (i=0;i<OnceCount;i++) {
            OnceNTF[PackedOnceList[i]]=eda_define_ntf(OnceHandle,NodeName,
                                        OnceTL[PackedOnceList[i]],OnceFL[PackedOnceList[i]],0);
            if (OnceNTF[PackedOnceList[i]]==BAD_THANDLE) {
                printf("ToFix: Bad handle %s.%s\n",OnceTL[PackedOnceList[i]],OnceFL[PackedOnceList[i]]);
            }
        }

        eda_lookup(OnceHandle);
        eda_wait(OnceHandle);

        for (i=0;i<OnceCount;i++) {
            TempValue=OnceValue[PackedOnceList[i]];
            err=eda_set_float(OnceHandle,OnceNTF[PackedOnceList[i]],&TempValue);
        }

        eda_write(OnceHandle);
        eda_wait(OnceHandle);

        for (i=0;i<OnceCount;i++) {
            err=eda_get_error(OnceHandle,OnceNTF[PackedOnceList[i]]);
            if (err!=FE_OK) {
                NlsGetText(err,msg,256);
                printf("ToFix: Default value tag %s.%s error\n%s\n",
                                    OnceTL[PackedOnceList[i]],OnceFL[PackedOnceList[i]],msg);
            } else OnceDone++;
        }

        eda_delete_group(OnceHandle);

        for (i=0;i<OnceCount;i++) {
            y[PackedOnceList[i]]=OnceValue[PackedOnceList[i]];
        }
    }
    printf("Default tags/values read: %i\tTags/values written: %i\n",OnceCount,OnceDone);

// debug display
/*
    printf("\ni\tPackedOnceList\tTag\t\tField\tValue\n");
    for (i=0;i<OnceCount;i++) {
            printf("%i\t%i\t\t%s\t%s\t%.2f\n",i,PackedOnceList[i],OnceTL[PackedOnceList[i]]
                                        ,OnceFL[PackedOnceList[i]],OnceValue[PackedOnceList[i]]);
    }
*/
```

```
// debug display
/*
    printf("\ni\tPackedNTFList\tTag\tField\n");
    for (i=0;i<NTFCount;i++) {
            printf("%i\t%i\t\t%s\t%s\n",i,PackedNTFList[i],TagList[PackedNTFList[i]],
                                                              FieldList[PackedNTFList[i]]);
    }
*/
}
#endif /* MDL_START */

/* Function: mdlOutputs =======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
}

/* Function: mdlTerminate =====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlInitializeConditions, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
}
```

SRT_main.c Source Code

```
#define VERSION   3.01

// ****************************
// New definitions and includes

// Event Actions - used for LOGFILE_MODE and CONSOLE_MODE
#define EA_ERRORS       (0)             // action only on errors
#define EA_ALL          (1)             // action on all steps

#define NO              (0)             // used for config stuff
#define YES             (1)             // ditto

#define LF_NAME_LENGTH  (40)        // length of the logfile name
struct CFG_STRUCT {
    char LogFileName[LF_NAME_LENGTH];       // name of the logfile
    int LogFile;                            // Y/N flag: write a logfile or not
    int LogFileClear;                       // Y/N flag: clear logfile before writing
    int LogFileConfig;                      // Y/N flag: write configuration to logfile
    int LogFileMode;                        // EA setting
    int ConsoleMode;                        // EA setting
    int ConsoleConfig;                      // Y/N flag: display config settings at start
    int SimStartPause;                      // Y/N flag: wait for keypress before beginning
    int SimReportStep;                      // positive integer, for report interval
} Cfg = {"logfile.txt",YES,NO,YES,EA_ALL,EA_ALL,YES,YES,10}; // defaults - if not set in CONFIG_FILE, left
        as is

#define CONFIG_FILE     "srt.cfg"
#include "string.h"                         // used for string parsing from configuration file
#include "time.h"                           // used for timing of steps, calculation of delay length
#include "conio.h"                          // used for keyboard exit check

#define WINAPI          __stdcall
#define DWORD           unsigned long
void                    WINAPI Sleep(DWORD dwMilliseconds);

#define CTRLX           (24)

 .
 .
 .
```

```
/* Function: main ================================================================
 *
 * Abstract:
 *      Execute model on a generic target such as a workstation.
 *
 *      usage: model [finaltime]
 *
 */
int_T main(int_T argc, char_T *argv[])
{
    SimStruct  *S;
    const char *status;
    char_T      str2[2];
    real_T      finaltime = 0.0;
    int_T       port      = 17725;
    double      tmpDouble; /* for parsing the comand line final time arg */

    int              LogActive=0,ModelLoop;
    float            StepSleepPerc=0;
    clock_t          StepSize,SampleSize,StepStartTime,StepEndTime,StepLength,StepDelay,StepOverrun;
    char             LogStartTime[9],LogStartDate[9],OverrunTime[9];
    unsigned long    ErrorCount=0,StepCount=0;

    char             StringIn[100],*TokenStr,*ParamStr,*SettingStr;
    FILE             *LogStream,*CfgStream;
    unsigned int     LineCount=0;

.
.
.

    /*******************************************************************
     Stuff to be done once, at the beginning of running the simulation
    *******************************************************************/

    // ********************************************* misc stuff
    SampleSize=(clock_t)(ssGetStepSize(S)*CLOCKS_PER_SEC);  // sample speed of the model
    StepSize=SampleSize*Cfg.SimReportStep;               // set (desired) step size in ticks
    printf("\nSRTLink OS: Version %.2f\n",VERSION);
    printf("Model: %s\n",ssGetModelName(S));
    printf("Sample interval (ms): %i\n",SampleSize);
    printf("Report interval (ms): %i\n\n",StepSize);

    // ********************************************* config file
    if ((CfgStream=fopen(CONFIG_FILE,"r"))==NULL) {
        printf("Unable to open config file '%s'\n",CONFIG_FILE);

        /* config file cannot be opened, start simulation with default settings, and
           create a new config file called CONFIG_FILE, with the settings in it, and a comment header */
        if ((CfgStream=fopen(CONFIG_FILE,"w"))==NULL) {
            printf("Unable to create new default config file, simulation will run with defaults\n");
        } else {
            // print default header to the config file
            // NB: justification problems in the parameter list are nominal fudges :)
            _strdate(LogStartDate);
            fprintf(CfgStream,"; Default configuration file created by SRTLink OS Version
        %.2f\n",VERSION);
            fprintf(CfgStream,"; Created on %s\n",LogStartDate);
            fprintf(CfgStream,";\n");
            fprintf(CfgStream,"; This file can be used to configure the operation of the SRT kernel\n");
            fprintf(CfgStream,"; by changing the value of the settings below.  Anything after a ; on\n");
            fprintf(CfgStream,"; a line is considered a comment and ignored.\n");
            fprintf(CfgStream,";\n");
            fprintf(CfgStream,"; LogFileName      - name of the logfile\n");
            fprintf(CfgStream,"; LogFile     - Y/N flag: write a logfile or not\n");
            fprintf(CfgStream,"; LogFileClear          - Y/N flag: clear logfile before writing\n");
            fprintf(CfgStream,"; LogFileConfig         - Y/N flag: write configuration to logfile\n");
            fprintf(CfgStream,"; LogFileMode       - EA setting\n");
            fprintf(CfgStream,"; ConsoleMode       - EA setting\n");
            fprintf(CfgStream,"; ConsoleConfig         - Y/N flag: display config settings at start\n");
            fprintf(CfgStream,"; SimStartPause         - Y/N flag: wait for keypress before beginning\n");
            fprintf(CfgStream,"; SimReportStep         - positive integer");
            fprintf(CfgStream,";\n");
            fprintf(CfgStream,"; An EA setting is either:\n");
            fprintf(CfgStream,"; 'Errors'    - take action on errors only\n");
            fprintf(CfgStream,"; 'All'       - take action on all events\n");
            fprintf(CfgStream,"; This applies to the LogFileMode, and the ConsoleMode.\n");
            fprintf(CfgStream,";\n");

            fprintf(CfgStream,"\n; Logfile settings\n");
            fprintf(CfgStream,"LogFileName\t= ");
            fprintf(CfgStream,"%s\n",Cfg.LogFileName);

            fprintf(CfgStream,"LogFile\t\t= ");
            if (Cfg.LogFile==YES) fprintf(CfgStream,"Yes\n");
                else fprintf(CfgStream,"No\n");

            fprintf(CfgStream,"LogFileClear\t= ");
            if (Cfg.LogFileClear==YES) fprintf(CfgStream,"Yes\n");
                else fprintf(CfgStream,"No\n");

            fprintf(CfgStream,"LogFileConfig\t= ");
            if (Cfg.LogFileConfig==YES) fprintf(CfgStream,"Yes\n");
                else fprintf(CfgStream,"No\n");

            fprintf(CfgStream,"LogFileMode\t= ");
            if (Cfg.LogFileMode==EA_ALL) fprintf(CfgStream,"All\n");
                else fprintf(CfgStream,"Errors\n");

            fprintf(CfgStream,"\n; Console settings\n");
            fprintf(CfgStream,"ConsoleMode\t= ");
            if (Cfg.ConsoleMode==EA_ALL) fprintf(CfgStream,"All\n");
                else fprintf(CfgStream,"Errors\n");

            fprintf(CfgStream,"ConsoleConfig\t= ");
            if (Cfg.ConsoleConfig==YES) fprintf(CfgStream,"Yes\n");
                else fprintf(CfgStream,"No\n");

            fprintf(CfgStream,"\n; Simulation settings\n");
            fprintf(CfgStream,"SimStartPause\t= ");
            if (Cfg.SimStartPause==YES) fprintf(CfgStream,"Yes\n");
                else fprintf(CfgStream,"No\n");

            fprintf(CfgStream,"SimReportStep\t= ");
            fprintf(CfgStream,"%i\n",Cfg.SimReportStep);

            fprintf(CfgStream,"\n; End of configuration file");
            fclose(CfgStream);
            printf("New default configuration file '%s' created\n",CONFIG_FILE);
        }
    } else {
        printf("Config file '%s' opened\n",CONFIG_FILE);
        while (!feof(CfgStream)) {
            if (fgets(StringIn,100, CfgStream)==NULL) {
                LineCount++;
                if (!feof(CfgStream)) {
                    printf("Config file error line %i: Undetermined\n",LineCount);
                }
            } else { // parse the token
                LineCount++;
                if (strncmp(StringIn,";",1)!=0) { // if the line is not a comment then parse

                    TokenStr=strtok(StringIn,";"); // cut out comments anywhere else in line
                    TokenStr=_strlwr(TokenStr);

                    if (TokenStr!=NULL) ParamStr=strtok(TokenStr," =\t\n;"); // find the parameter
                    if (ParamStr!=NULL)     SettingStr=strtok(NULL," =\t\n;"); // find the setting

                    // printf("|%s| = |%s|\n",ParamStr,SettingStr); // token display debug stuff

                    if ((ParamStr!=NULL) && (SettingStr!=NULL)) {

                        // log file name
                        if (!strcmp(ParamStr,"logfilename")) {
                            // no checking is done for the validity of the file name, truncated to
                                                                                        LF_NAME_LENGTH
                            strncpy(Cfg.LogFileName,SettingStr,LF_NAME_LENGTH);
                        }

                        // log file active
                        if (!strcmp(ParamStr,"logfile")) {
```

```
            if (!strcmp(SettingStr,"yes")) Cfg.LogFile=YES; else
                if (!strcmp(SettingStr,"no")) Cfg.LogFile=NO; else {
                    printf("Config file error line %i: ",LineCount);
                    printf("LogFile must be Yes or No\n");

                }
        }

        // clear logfile before writing
        if (!strcmp(ParamStr,"logfileclear")) {
            if (!strcmp(SettingStr,"yes")) Cfg.LogFileClear=YES; else
                if (!strcmp(SettingStr,"no")) Cfg.LogFileClear=NO; else {
                    printf("Config file error line %i: ",LineCount);
                    printf("LogFileClear must be Yes or No\n");

                }
        }

        // write config to logfile
        if (!strcmp(ParamStr,"logfileconfig")) {
            if (!strcmp(SettingStr,"yes")) Cfg.LogFileConfig=YES; else
                if (!strcmp(SettingStr,"no")) Cfg.LogFileConfig=NO; else {
                    printf("Config file error line %i: ",LineCount);
                    printf("LogFileConfig must be Yes or No\n");

                }
        }

        // logfile mode
        if (!strcmp(ParamStr,"logfilemode")) {
            if (!strcmp(SettingStr,"errors")) Cfg.LogFileMode=EA_ERRORS; else
                if (!strcmp(SettingStr,"all")) Cfg.LogFileMode=EA_ALL; else {
                    printf("Config file error line %i: ",LineCount);
                    printf("LogFileMode must be Errors or All\n");

                }
        }

        // console mode
        if (!strcmp(ParamStr,"consolemode")) {
            if (!strcmp(SettingStr,"errors")) Cfg.ConsoleMode=EA_ERRORS; else
                if (!strcmp(SettingStr,"all")) Cfg.ConsoleMode=EA_ALL; else {
                    printf("Config file error line %i: ",LineCount);
                    printf("ConsoleMode must be Errors or All\n");

                }
        }

        // write config to console
        if (!strcmp(ParamStr,"consoleconfig")) {
            if (!strcmp(SettingStr,"yes")) Cfg.ConsoleConfig=YES; else
                if (!strcmp(SettingStr,"no")) Cfg.ConsoleConfig=NO; else {
                    printf("Config file error line %i: ",LineCount);
                    printf("ConsoleConfig must be Yes or No\n");

                }
        }

        // pause for keypress at beginning of simulation
        if (!strcmp(ParamStr,"simstartpause")) {
            if (!strcmp(SettingStr,"yes")) Cfg.SimStartPause=YES; else
                if (!strcmp(SettingStr,"no")) Cfg.SimStartPause=NO; else {
                    printf("Config file error line %i: ",LineCount);
                    printf("SimStartPause must be Yes or No\n");

                }
        }

        // number of steps between reporting to logfile/console
        if (!strcmp(ParamStr,"simreportstep")) {
            // no range checking is done for this value, although only the
            // magnitude of negative numbers is used, sign is ignored
            Cfg.SimReportStep=(int)floor(abs(atoi(SettingStr)));
        }

    } // end of null check parse
```

```
                } // end of token parsing
            } // end of check for fgets error
        } // end of file
        fclose(CfgStream);
    }

    // ********************************************** print config to console
    if (Cfg.ConsoleConfig) {
            printf("Logfile Name:\t\t\t");
            printf("%s\n",Cfg.LogFileName);

            printf("Logfile active:\t\t\t");
            if (Cfg.LogFile==YES) printf("Yes\n");
                else printf("No\n");

            printf("Clear file before logging:\t");
            if (Cfg.LogFileClear==YES) printf("Yes\n");
                else printf("No\n");

            printf("Write config to file:\t\t");
            if (Cfg.LogFileConfig==YES) printf("Yes\n");
                else printf("No\n");

            printf("Logfile Mode:\t\t\t");
            if (Cfg.LogFileMode==EA_ALL) printf("All events\n");
                else printf("Errors only\n");

            printf("Console Mode:\t\t\t");
            if (Cfg.ConsoleMode==EA_ALL) printf("All events\n");
                else printf("Errors only\n");

            printf("Write Config to Console:\t");
            if (Cfg.ConsoleConfig==YES) printf("Yes\n");
                else printf("No\n");

            printf("Pause before simulating:\t");
            if (Cfg.SimStartPause==YES) printf("Yes\n");
                else printf("No\n");

            printf("Report interval (steps):\t");
            printf("%i\n",Cfg.SimReportStep);
    }

    // ********************************************** setup logfile
    if (Cfg.LogFile) {
        if (Cfg.LogFileClear) { // overwrite any existing logfile
            if ((LogStream=fopen(Cfg.LogFileName,"w"))==NULL) {
                printf("\nCan't create file '%s' - No events will be logged\n\n",Cfg.LogFileName);
                LogActive=0;
            } else {
                printf("\nNew logfile '%s' opened for writing\n\n",Cfg.LogFileName);
                LogActive=1;
            }
        } else { // open a file if it exists, or create a new one
            if ((LogStream=fopen(Cfg.LogFileName,"a+"))==NULL) {
                printf("\nCan't open or create file '%s' - No events will be logged\n\n",Cfg.LogFileName);
                LogActive=0;
            } else {
                printf("\nLogfile '%s' opened for appending\n\n",Cfg.LogFileName);
                LogActive=1;
            }
        }
    } else { LogActive=0; }

    if (LogActive==1) { // if a file has been opened, write logfile header
        setvbuf(LogStream,NULL,_IONBF,0);

        // primary header
        fprintf(LogStream,"\nSRTLink OS: Version %.2f\n",VERSION);
        fprintf(LogStream,"Model: %s\n",ssGetModelName(S));
        fprintf(LogStream,"Sample interval (ms): %i\n",SampleSize);
        fprintf(LogStream,"Report interval (ms): %i\n\n",StepSize);

        // optional header
        if (Cfg.LogFileConfig==YES) {
            fprintf(LogStream,"Logfile Name:\t\t\t");
            fprintf(LogStream,"%s\n",Cfg.LogFileName);

            fprintf(LogStream,"Logfile active:\t\t\t");
            if (Cfg.LogFile==YES) fprintf(LogStream,"Yes\n");
                else fprintf(LogStream,"No\n");

            fprintf(LogStream,"Clear file before logging:\t");
            if (Cfg.LogFileClear==YES) fprintf(LogStream,"Yes\n");
                else fprintf(LogStream,"No\n");

            fprintf(LogStream,"Write config to file:\t\t");
            if (Cfg.LogFileConfig==YES) fprintf(LogStream,"Yes\n");
                else fprintf(LogStream,"No\n");

            fprintf(LogStream,"Logfile Mode:\t\t\t");
            if (Cfg.LogFileMode==EA_ALL) fprintf(LogStream,"All events\n");
                else fprintf(LogStream,"Errors only\n");

            fprintf(LogStream,"Console Mode:\t\t\t");
            if (Cfg.ConsoleMode==EA_ALL) fprintf(LogStream,"All events\n");
                else fprintf(LogStream,"Errors only\n");

            fprintf(LogStream,"Write Config to Console:\t");
            if (Cfg.ConsoleConfig==YES) fprintf(LogStream,"Yes\n");
                else fprintf(LogStream,"No\n");

            fprintf(LogStream,"Pause before simulating:\t");
            if (Cfg.SimStartPause==YES) fprintf(LogStream,"Yes\n");
                else fprintf(LogStream,"No\n");

            fprintf(LogStream,"Report interval (steps):\t");
            fprintf(LogStream,"%i\n",Cfg.SimReportStep);
        }
}

if (Cfg.SimStartPause) {
    printf("Press a key to start simulation...\n");
    while (!_kbhit()) {}
}
printf("Press Ctrl-X to stop simulation at any time\n\n");

// inserted here to take the time at the beginning of the simulation
_strdate(LogStartDate);
_strtime(LogStartTime);
if (LogActive) fprintf(LogStream,"Log started %s, %s\n",LogStartTime,LogStartDate);

// ********************************************** simulation loop
while (!GBLbuf.stopExecutionFlag) {
    StepStartTime=clock();
    for (ModelLoop=1;ModelLoop<=Cfg.SimReportStep;ModelLoop++) {
        rt_OneStep(S);
    }
    StepEndTime=clock();

    StepCount++;

    /* Through the logfile/handler code, StepLength is just the time taken for the model
       itself, not including the overhead code.  At the end of the logfile/handler code, the
       time is found again, and the sleep delay worked out from this.
    */
    StepLength=StepEndTime-StepStartTime;

    if (StepLength<StepSize) { // model has run inside sampletime
        StepSleepPerc=100-((float)StepLength/(float)StepSize)*100;

        if (Cfg.ConsoleMode==EA_ALL) {
            printf("%% of step sleeping:\t %.1f%%\n",StepSleepPerc);
            printf("Ctrl-X to stop simulation\n\n");
        }

        if (LogActive==1 && Cfg.LogFileMode==EA_ALL) {
            fprintf(LogStream,"Simulating: %ims,\t Sleeping: %.1f%%\n",StepLength,StepSleepPerc);
        }

        // this is done again, here, to take into account time to execute the SRT
        // overhead code
```

```
            StepEndTime=clock();
            StepLength=StepEndTime-StepStartTime;
            StepDelay=((StepSize-StepLength)*CLOCKS_PER_SEC)/1000;
        } else { // model has overrun sample time
            ErrorCount++;

            StepOverrun=StepLength-StepSize;
            _strtime(OverrunTime);

            printf("** Overrun: %ims (%.1f%%) at %s\n",StepOverrun,
                                                    ((float)StepOverrun/(float)StepSize)*100,OverrunTime);
            printf("Ctrl-X to stop simulation\n\n");

            if (LogActive==1) {
                fprintf(LogStream,"** Simulating: %ims,\t Overrun: %i (%.1f%%) at %s\n",StepLength,
                                    StepOverrun,((float)StepOverrun/(float)StepSize)*100,OverrunTime);
            }

            // time overflowed, so no delay necessary
            StepDelay=(clock_t)0;
        }

        Sleep((DWORD)StepDelay);

        if (_kbhit()) {
            if (_getch()==CTRLX) {
                GBLbuf.stopExecutionFlag=1;
            }
        }

        if (ssGetStopRequested(S)) break;
    }

    if (!GBLbuf.stopExecutionFlag && !ssGetStopRequested(S)) {
        /* Execute model last time step */
        StepCount++;
        rt_OneStep(S);
    }

    /********************
     * Cleanup and exit *
     ********************/

    _strtime(LogStartTime); // variables are reused here as temp store for logfile output
    _strdate(LogStartDate);
    if (LogActive) {
        fprintf(LogStream,"Simulation finished %s, %s\n",LogStartTime,LogStartDate);
        fprintf(LogStream,"Error count for simulation: %i of %i steps\n",ErrorCount,StepCount);
        fclose(LogStream);
    }
.
.
.

/* End of Main.c */
```

APPENDIX B: SIMEVAP USER MANUAL

Table of Contents

Installation and operation

Installation of Fix32

Before installation of the SimEvap software, Fix32 must be installed on the machine as a SCADA node to run the simulator. For details of how to do this, refer to the *Fix32 System Setup* manual which accompanies Fix32. For the purposes of this installation guide, it is assumed that Fix32 has been installed in the default directory of C:\Fix32.

Installation of SimEvap

Once Fix32 is installed, the components for SimEvap must be installed. This is done by copying the contents of the "Fix32" directory (4 folders and their contents, named "Local", "Pdb", "SimEvap" and "SimPics") on the SRTLink/SimEvap CD into the "Fix32" directory on the hard drive. All files that already exist on the hard drive should be replaced by those from the CD.

Once the folders are copied onto the hard drive, Fix32 must be configured to use the correct directories and options. This is run by running the *Fix System Configuration* utility, and selecting "Open" from the File menu. From the dialog box, select "Model.SCU" and open it. This will make the settings contained in the file current, and set the "Model.SCU" file as default.

The "SimEvap" folder also contains a shortcut for SimEvap. This shortcut should be copied into a directory where it can be accessed to run the simulator whenever needed. The "Intellution FIX" folder in the Windows Start menu is ideal for this, allowing a user to run Fix32 and then run the simulator.

Note: This shortcut assumes that the application resides in the C:\Fix32\SimEvap directory. If Fix32 has been installed into another directory or drive then a new shortcut needs to be created to point at the new location of "SimEvap.exe".

Running the Simulator

Once installed as described above, running the simulator consists of running Fix32 as usual. Once active, selecting the "SimEvap" shortcut from the "Intellution FIX" folder in the Windows Start menu will activate the simulator.

If at any stage the simulator becomes unstable or the user wishes to restart the simulation, it can be shut down by going to the SimEvap program in the Windows Task Bar and pressing Ctrl-X. This will end the current simulation, and SimEvap can then be executed again from the "Intellution FIX" folder.

Simulator restrictions and capabilities

Not all of the controls normally available to the operators are active in the simulator, due to limitations in the model. These limitations apply to:

- Pumps and valves. Although visually the pumps are active and the required valves are open or closed, altering their state makes no changes to the operation of the model. For the purposes of the current simulator, they are effectively cosmetic.

- MVR fan. The simulation of the electrical components of the MVR fan is not part of the model. Therefore the MVR fan page has no values available to the operator.

It should be noted that it is possible to push the model beyond it's capabilities but that it will still generate temperatures and flows in this region. These numbers will not be representative of the operation of the true evaporator. However, in pushing the model into this state, an operator if running the real plant would have destroyed it by that stage in any case.

Alterations to the setpoints of the control loops are valid, as is the switching from automatic to manual and back, along with the manual setting of the valve position. Alterations to the PID controller values (the proportional, integral and derivative values) are also valid and will affect the model, although due to the limitations of modelling any process of this complexity changing the PID controller values on the model does not reflect the operation of the plant with the same change. Any such alteration will alter the response of the model in the same general fashion as it would on the plant; making changes to the plant controller loops based on changes tested on the model is not recommended.

SimEvap Controls Page

The SimEvap Controls Page button can be found on the main selection bar, at the right hand end (Figure 26). The text on the button is red in order to differentiate it from the usual buttons in the control system. The control page can be accessed at any time and any changes made have instant effect on the simulation.

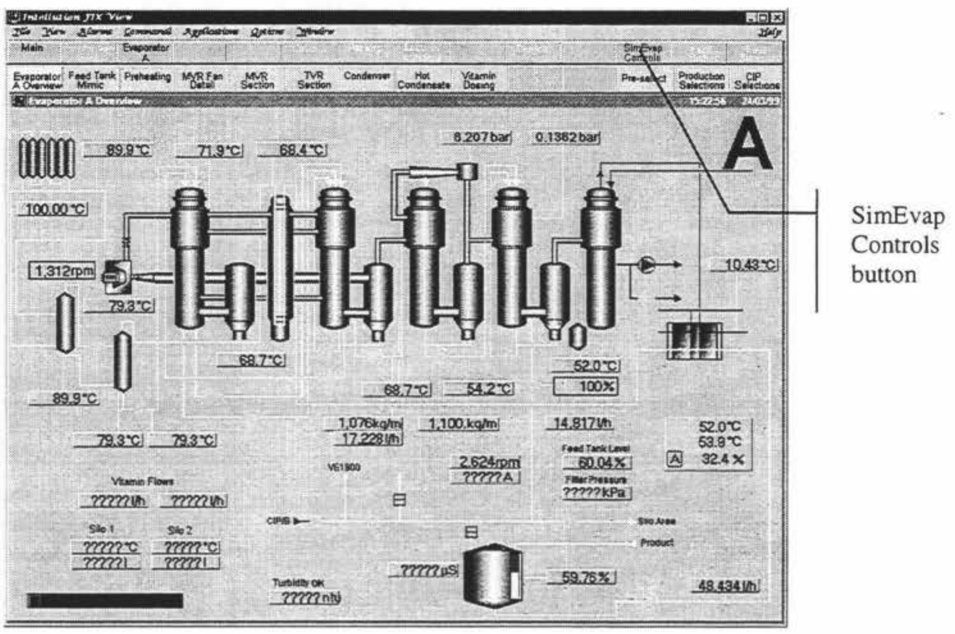


Figure 26 Location of the SimEvap Controls Page selection button

The SimEvap Controls Page (Figure 27) itself is laid out in a similar fashion to the Evaporator Overview page. Variables that can be altered are displayed in dialog boxes around the evaporator and are altered by double-clicking on the variable desired, before entering a new value. The holding tube time is the only "constant" variable; the other variables all have nominal values, along with frequency and time constants.

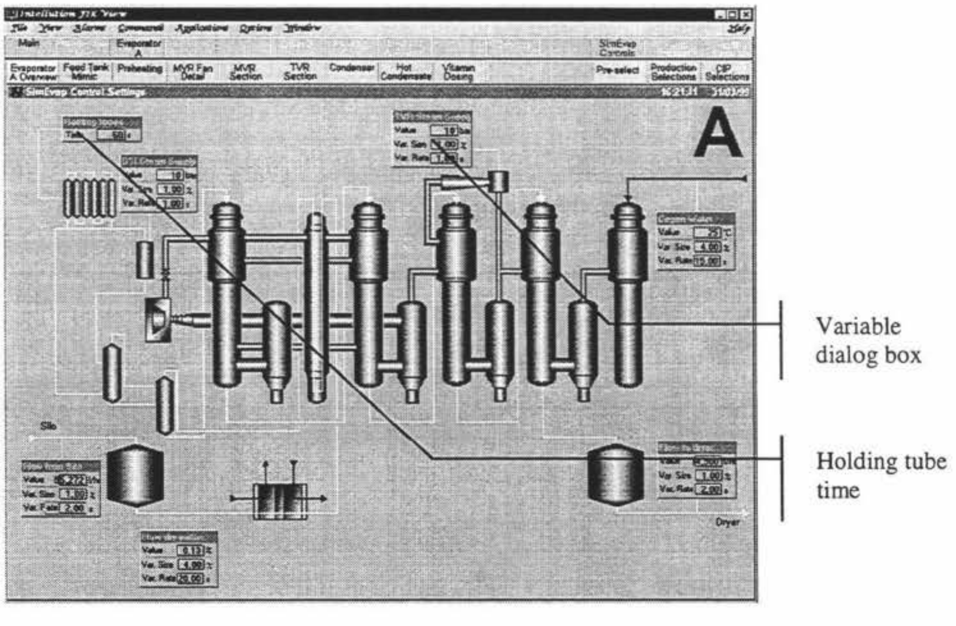
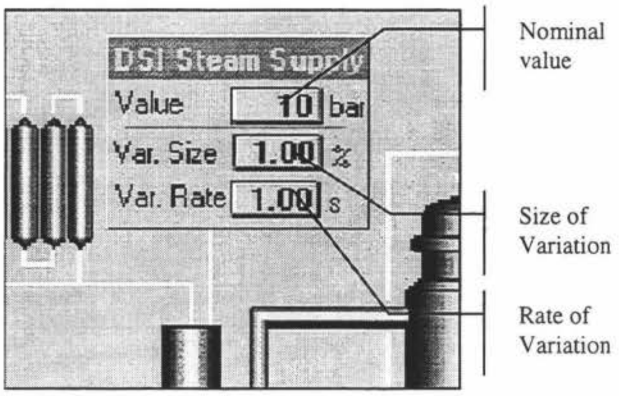


Figure 27 SimEvap Controls page

A close up of one of the variable dialog boxes can be seen in Figure 28. In common with all of the variable dialogs, the top value is the nominal value (in this case measured in bar). The bottom two control the variation in the value, with the percentage size controlling the amplitude of the variation (in Figure 28 this is 1% of 10 bar, or 0.1 bar), and the variation rate controlling the speed which the value can change at (set to 1.0 seconds). This equates to a steam pressure which is nominally at 10 bar ± 0.1 bar, with the fastest possible change (ie. from 9.9 bar to 10.1 bar) being no faster than 1 second.


DSI Steam Supply

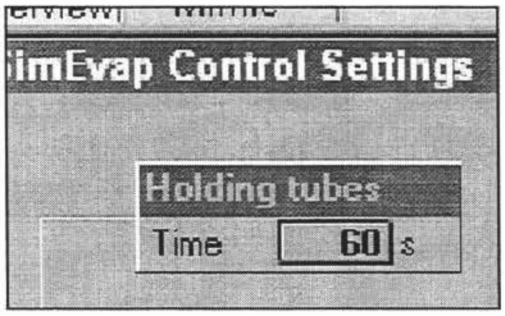
Value 10 bar

Var. Size 1.00 %

Var. Rate 1.00 s

Figure 28 Close up of SimEvap control screen variable dialog box

All of the variables accessible from the SimEvap Controls Page are shown in the same fashion, with the exception of the Holding Tube time. The dialog box which is used to specify this is shown in Figure 29. As can be seen, there is no variation size or rate, as would be expected. Note that any changes made to the holding tube time will take the same amount of time before the new delay is fully active. For example, if the holding tube time is altered from 60 seconds to 120 seconds, then 120 seconds must elapse before the "holding tubes" in the model are working with a full 120 second delay. In practice, this effect will normally be unnoticeable.


imEvap Control Settings

Holding tubes

Time 60 s

Figure 29 Close up of the holding tube dialog box

Apart from the holding tube time, the variables that are available to be set are:

- DSI steam pressure
- TVR steam pressure
- Temperature of water from the Cogeneration plant
- Flow from the concentrate tank to the dryer
- Flow of milk from the silos to the feed tank
- Dry matter of the milk flow from the silos

All of these variables are configured as discussed above with a size and rate of variation. If no variation in a variable is desired, set the variation size percentage to zero.

Note that these variables are not usually controllable. They are available in order to set up particular situations such as the DSI pressure dropping, or to show the effect of a high dry matter on the operation of the evaporator. They could be used for control purposes (ie. the evaporator is heating up, so lower the temperature of the cooling water supply from the Cogeneration plant), but doing this achieves nothing with respect to training an operator – would they respond in this fashion to the real evaporator overheating?

Technical Addendum

The file "SRT.cfg" in the Fix32\SimEvap directory contains the configuration settings for the simulator application. These settings can be altered in order to log the performance of the simulator to disk, and to set the display options for the simulator task window. For a description of these options, refer to SRT Configuration File Options in Appendix C: SRT User Manual. Note that this configuration file does not include the values displayed on the model control page.

The start up values for the model control variables are specified in the "InitFix.cfg" file, used by SimEvap. They can be altered, but it should be noted that the model is designed to start up in steady state with these values. If they are changed, then the model will not start in steady state (although it will still settle to steady state). Alterations to the default values for the PID controllers can also be made in the "InitFix.cfg" file, but again this will disturb the start-up state of the model.

For a list of the tags concerning the model control variables, refer to Appendix E : Fix32 Database and Control Screen Modifications.

APPENDIX C: SRT USER MANUAL

Table of Contents

Installation of the Fix/Visual C/Simulink Development Environment

Introduction

For development of this system, three separate pieces of software need to be installed: Fix32 6.0 or above, with the EDA libraries and the control database to be used with the simulator, Microsoft Visual C++ 5.0, and Matlab, with the Real-time Workshop. In synergy these applications create the development environment for the simulator.

Ideally, a fresh install of Windows NT should be made before installing the above three software packages. This may not be strictly necessary, but if for an unknown reason the installed software does not work in conjunction, then reinstalling Windows NT could solve the problem.

The installation order of the software does not matter, but the order that they are presented in is the order that they were installed in for the development of this simulator. However, Microsoft Visual C++ should be installed before Matlab is configured to use an external compiler.

Fix32

Fix should be installed to the default directory C:\Fix32, and should also be configured as a SCADA node. If it is not configured as a SCADA node, then the computer will only be able to view other nodes, not manipulate a control database of its own. The name of the node (set in the Fix32 Setup program, or in the SCU configuration file) should be unique, if the computer is going to be attached to other Fix SCADA nodes. This is to ensure that reads and writes with the EDA library do not interfere with other nodes.

Any of the other options are up to the requirements of the user, and any other demands that are to be made on the control system. View and Draw are installed by default, and none of the other options are necessary for the operation of the simulator (although they might be necessary for the control system concerned to operate).

Accompanying Fix is the EDA disk. Once Fix has been installed, the EDA libraries need to be installed as well. C:\Fixtools\ is the default directory for the libraries, the help, and the examples. The EDA install program gives the user the option to install for Visual C, Visual Basic, or both – only the required development environment is needed (Visual C in the case of SRTLink).

Microsoft Visual C++ 5.0

This software needs to be installed with the default options. Any of the optional libraries need only be installed subject to the requirements of the software to be written. Note that none of the extended libraries are necessary for compilation of the SRTLink software.

Simulink

Matlab needs only to be installed with Simulink and the RTW; installation of any of the other toolboxes is subject to the requirements of the simulation being developed. Once installed, the options for creating C MEX files must be set. This is done by typing "mex –setup", and answering the questions asked. This process creates a default configuration file in the \matlab\bin directory called "mexopts.bat". This file needs some manual modifications made in order to find and use the EDA libraries.

Use Notepad to open mexopts.bat, and under the "General Parameters" section, add "C:\Fixtools\Include" to the INCLUDE parameter. Also add "C:\Fixtools\Lib" to the LIB parameter. These parameters tell Matlab where to look for the EDA include files, and the EDA libraries compiled into the mexfile.

Under the "Linker Parameters" section, add to the LINKFLAGS parameter both "fixtools.lib" and "dmacsdba.lib". These are the static libraries that contain the EDA calls. Once included in the "mexopts.bat" file in this fashion, whenever a mexfile is created at the Matlab command line these libraries will be included in the link by the MSVC linker. The source code still needs to include "fixtools.h" and "fixeda.h" in order to utilise the libraries, but due to the modifications in the "General Parameters" section the linker will know where these include files are located.

Installing the Simulink Components of SRTLink

Before using the SRTLink software to create a simulator, the CMEX files for each driver and the Simulink library for inserting the drivers into a model must be installed. These files can be installed anywhere under the \matlab directory, providing that the directory that they are copied into is added to the Matlab path. The suggested directory for these components is \matlab\edalink.

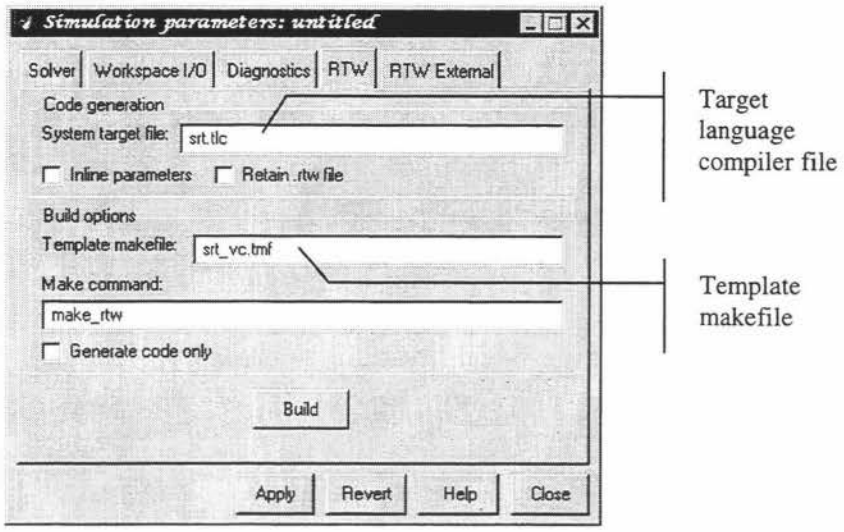
Create the directory (this manual will assume \matlab\edalink), and copy into it InitFix.dll, FromFix.dll, ToFix.dll, and EDALink.mdl. Start Matlab, and under File go to Set Path. Within the dialog box that is brought up go to Add to Path under the Path pulldown menu, and type in c:\matlab\edalink, and click on OK. This will add the directory to the Matlab path, and give Simulink access to both the EDALink library and the driver CMEX files.

Adding the SRT files to the RTW

Once Matlab/Simulink has been installed with the RTW, the configuration files for SRTLink need to be installed in the \matlab\rtw\c\grt directory. These files are "srt_vc.tmf", "srt.tlc", and "srt_main.c". Refer to the RTW Support Files section for details on the use and purpose of each of the files.

RTW Compilation Options for SRTLink

In the options for a Simulink model is a tab page devoted to options for the RTW (Figure 30). For the model to be compiled with the SRTLink manager software, two options must be changed. The system target file at the top of the window should read "srt.tlc", and the template make file option should read "srt_vc.tmf".


Simulation parameters: untitled

Solver | Workspace I/O | Diagnostics | RTW | RTW External

Code generation

System target file: srt.tlc

☐ Inline parameters ☐ Retain .rtw file

Build options

Template makefile: srt_vc.tmf

Make command: make_rtw

☐ Generate code only

Figure 30 RTW options screen, set up for SRTLink

Options, such as static libraries to include in the linking of the software are set within the files specified in this dialog box. For the details of these options, refer to the RTW Support Files section.

In general, the RTW does not support compilation of models using variable step solvers. This means that the solver type (set in the Solver tab page) must be set to Fixed Step. The step size should also be set to a particular value as opposed to "auto". This ensures that the permissible sample intervals for the drivers is known, and not dynamic between simulations.

RTW Support Files

Srt_vc.tmf

This file is the template makefile used by the RTW to create the C code, and to compile the generated code into an application. It is within this file that the name of the C code kernel used to create the application is set (in this case, "srt_main.c"). It also specifies search paths for external resources, and the names of external libraries to be linked into the application. Note that these paths and libraries are separate (but may be identical) to those specified in mexopts.bat for the creation of CMEX files.

In order to use a particular TMF file for a given model, the name of the TMF file needs to be set in the template makefile field of the RTW tab in the model parameters. Once this is set, then a build of the model will use the settings specified in the TMF file.

The file "srt_vc.tmf" is based on "grt_vc.tmf" supplied with Matlab. Modifications have been made to include the EDA static libraries in the linking of the software. Note that any static libraries that are used in S-functions must be included here as well for the S-functions to work properly. For a full list of modifications made to "srt_vc.tmf", refer to the revision history at the beginning of the file.

Mexopts.bat

Contained within this file are the settings used for the creation of CMEX files, for use in S-function blocks in Simulink. Note that these settings are separate from those in the "srt_vc.tmf" file, although the paths and libraries concerned may be identical. The parameters set here refer only to compiling CMEX files from the command line with the "mex" command – if the model is to be compiled with the RTW after development and testing, then the library paths and names also need to be included "srt_vc.tmf".

Srt.tlc

With respect to the SRTLink software, this file is only used to enable or disable the creation of a data log in the form of a .MAT file. This is done by changing the `%assign MatFileLogging = 1` line to `%assign MatFileLogging = 0`. Making this change to "srt.tlc" stops the generated EXE file from creating the data log.

For a full discussion of the options that can be set in a .TLC file, refer to the *Simulink TLC Reference* supplied with Matlab.

EDALink

EDALink is a Simulink library created for the driver software (Figure 31). It contains the drivers and the DCP, the EDA/Sim switch and a unit conversion block that is useful for both incoming and outgoing values.

It is from this library that the drivers are dropped into a simulation model, and connected. Note that once the drivers have been inserted into a model, the library link must be broken. This can be done by selected one of the drivers, and going to "Break Library Link" in the Edit menu. This must be done singly for each of the drivers and the DCP. This ensures that any modifications made to the drivers are particular to that model, without affecting the EDALink library or others model with the drivers inserted into them.

To bring up the EDALink library for use, just type "edalink" at the Matlab command prompt. This assumes that the library and the drivers have been installed, as discussed in the "Installing the Simulink Components of SRTLink" section.

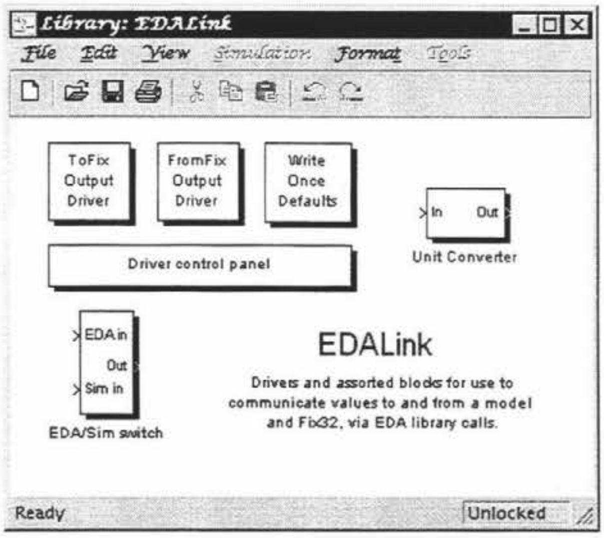


Figure 31 The EDALink library

Inserting the Drivers into a Model

To insert the drivers themselves into a model, all that is necessary is to drag them from the EDALink library (including the Driver Control Panel) into the top level of the model. This copies the S-functions, and the timing support logic inside the Driver Control Panel. Once this step is performed, the library link between the newly copied drivers and EDALink must be broken. Selecting one of the drivers or the control panel, and going to the Edit pulldown menu, and choosing "Break Library Link" does this. This must be done separately for each block.

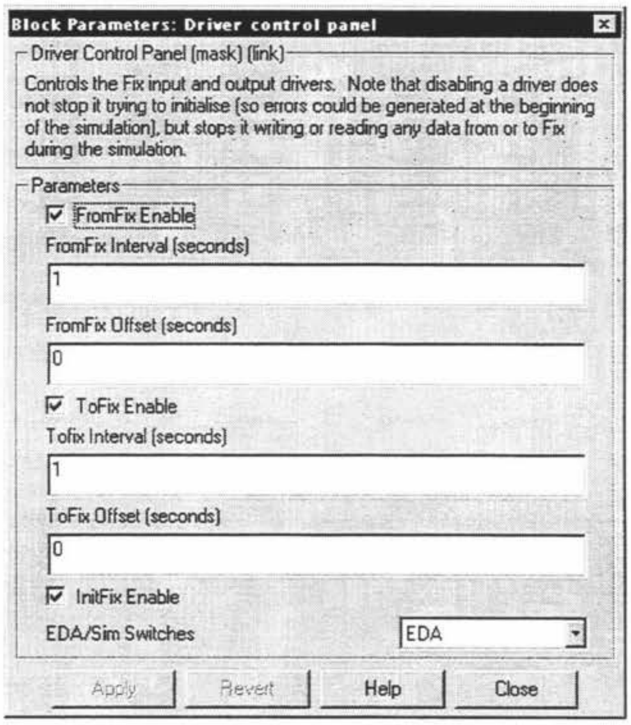
Once installed into a model, the drivers can be connected into the control points of the model via From/Goto blocks. Both FromFix and ToFix have arrays of From/Goto blocks set up according to the structure of the drivers so that they can just be copied from the driver blocks into the relevant point in the model. Note that once a Goto block from ToFix is inserted into the model, it must be deleted from within the driver subsystem (Goto blocks must have a unique label throughout the model). This detail does not apply to FromFix; copying a From block out of the FromFix subsystem is allowed (multiple From blocks can exist for a given Goto label). It is recommended that the From blocks are not deleted from the FromFix subsystem when inserted into the model; this ensures that if they are later removed from the model that Simulink will still have a From/Goto pair and will not generate warnings upon simulation.

Also laid out within the structure of the model are generic labels for each From/Goto pair. These labels are intended to reflect the configuration files of the drivers. In this fashion it can be seen which From/Goto blocks refer to which tag and field within the Fix32 control database.

Configuring the Drivers

The Driver Control Panel

The options for the drivers are set in the Driver Control Panel (DCP), shown in Figure 32. Each driver can be enabled or disabled, and in addition FromFix and ToFix have a sample interval and offset time, both measured in seconds. The sample time is the interval that the driver is activated to communicate with the control system, whereas the offset is the starting point of the sample intervals. For example, with the sample time set to a second, and the offset to half a second, the driver will trigger after half a second of simulation time, on one and a half seconds, two and a half, and so on. This function is useful if, due to speed restrictions, you wish to trigger the drivers at different times to smooth out the communication load.


Block Parameters: Driver control panel

Driver Control Panel (mask) (link)

Controls the Fix input and output drivers. Note that disabling a driver does not stop it trying to initialise (so errors could be generated at the beginning of the simulation), but stops it writing or reading any data from or to Fix during the simulation.

Parameters

[✓] FromFix Enable

FromFix Interval (seconds): 1

FromFix Offset (seconds): 0

[✓] ToFix Enable

Tofix Interval (seconds): 1

ToFix Offset (seconds): 0

[✓] InitFix Enable

EDA/Sim Switches: EDA

Figure 32 Driver Control Panel

Note that disabling a driver only disables it once the simulation has started – Simulink must of necessity initialise every block within a model. This means that the drivers will still read in their respective configuration files and try to connect with Fix32, but once Simulink starts stepping through the simulation no communication will occur.

The sample interval of each driver must be a multiple of the sample time of the model. This restriction also applies to the offset time. The sample time of the model can be set in the Simulink model Options in the Solver tab page. If this restriction is broken, then Simulink will not run or compile the model and will notify the user of the problem.

The Driver Configuration Files

Each of the drivers has a configuration file associated with it which contains a list of the tags and fields that each channel in the driver is linked with. The names of these files are "ToFix.cfg", "FromFix.cfg", and "InitFix.cfg", each referring to the named driver.

The "ToFix.cfg" file and the "FromFix.cfg" file both have a common format. Each line is in the form of "*x* = tag.field", where *x* refers to the number of the channel of the driver within Simulink. The tag and field specification refers to the location within Fix32 for either the storage or retrieval of data, dependant on which driver is under discussion.

The format of the "InitFix.cfg" file is the same as the previous two, with one small difference. At the end of each line a numerical value must be specified, separated from the rest of the line by a comma. For example,

```
12 = TANKSP.F_CV,20
```

specifies that upon initialisation of the model a value of 20 will be inserted into the TANKSP block within Fix32. The value of 20 will also be available within the model, if needed, on channel 12 of the InitFix driver.

The only other option which can be changed in the configuration files for the drivers is the name of the node that communication is carried out to. This is done by specifying "NODE = *xxxxxx*" in any of the three configuration files. Note that each driver uses an independent node; they are not required to write or read data from the same node (although in normal operation it would be expected that the node name would be the same for each driver). If a node is not specified within the configuration, a default node name of "MODEL" is used by the driver to build the NTF addresses.

All of the drivers will ignore any text occurring on a line following a semicolon, allowing the user to insert comments into the configuration file. For example:

```
; configuration file for the holding tank simulator
NODE = MODEL          ; nodename for the model
1    = TANK1.F_CV     ; this is the level of tank 1
2    = PUMP1.F_CV     ; this is the speed of pump 1
```

The list of channels and associated tags and fields also does not have to be in numerical order, and any invalid lines are reported to the screen and the line ignored. An invalid line can be caused by a non-existent tag or field being specified, or by giving a channel number than is larger than the maximum for the driver (300 for each of the drivers). An error is also generated by InitFix if no value is specified at the end of a line. If a channel number occurs more than once within a configuration file, then the most recent valid definition is used by the driver – the others are ignored.

SRT Configuration File Options

The file "SRT.cfg" is a text file read in by a simulator at runtime. The structure of the file is similar to that of a .INI file; the options do not need to be in any order, they are not case sensitive, and anything after a semicolon on a line is considered a comment and ignored. A given option is set with the option label, followed by an equals sign, followed by the value of the option. Any invalid options are reported by the simulator when run and then ignored (the default value is used in place of the invalid setting). All options are specified without quotes, although they are listed below within quotes. For example, `LogFile=Yes` is valid, while `LogFile="Yes"` is not.

The configurable options are:

- LogFileName The name of the log file to be written to disk. This option is moot if LogFile is set to No. The filename is specified without quotes with an extension if one is desired ie. `LogFileName=mylogfile.txt`. The default value for this option is "logfile.txt"
- LogFile Whether or not a log file is to be written to the disk. Valid settings for this option are "Yes" and "No". The default value for this option is "Yes".
- LogFileClear Whether the current simulation log is appended to an existing log, or an existing logs are cleared before the simulation begins. Valid settings are "Yes" and "No"; setting this option to "Yes" will clear an existing log. If no previous log

exists (called LogFileName), then this option has no effect. This option also has no effect if LogFile is set to "No". The default value for LogFileClear is "No" (the new log will be appended to the previous log).

- LogFileConfig Whether or not the current configuration settings of the simulator are written into the log file. Valid settings are "Yes" and "No". This option has no effect if LogFile is set to "No". The default value for LogFileConfig is "Yes".
- LogFileMode Whether all intervals are written to the log file, or only intervals in which an overrun has occurred. Valid settings are "All" (all intervals are written to the log file) and "Errors" (only intervals with errors are written to the log file). This option has no effect if LogFile is set to "No". The default value for LogFileMode is "All".
- ConsoleMode Whether all intervals, or just intervals in which errors occur are printed into the console window while the simulation is running. Valid settings are "All" (all intervals are displayed in the console window) and "Errors" (only intervals with errors are displayed). The default value for this option is "All".
- ConsoleConfig Whether or not the current configuration settings of the simulator are displayed in the console window when the simulator is started. Valid settings are "Yes" and "No". The default value for this option is "Yes".
- SimStartPause Whether or not the simulator waits for a keypress before starting the simulation. Valid settings are "Yes" and "No". If this option is set to "Yes", then once the simulation is initialised the simulator will display "Press a key to start simulation..." in the command window and wait until a key is pressed before beginning the simulation loop. The default value for this option is "Yes".
- SimReportStep The interval (in steps) in which a report is generated, either for a log file or for display in the console window. This must be a positive non-zero integer. Note that the amount of real time that this figure represents depends on the step size of the simulation. The default value for this option is 10.

Log file Format

The log file for a simulation is merely to keep a record of the simulator timing and overflows. It does not record any of the values or characteristics of the model over the period the simulator is active. To record the values within the model, the simulator needs to be recompiled with the `MatFileLogging = 1` option set in the "Srt.tlc" file.

An example of a typical log file is shown in Figure 33. The first four lines of the file are the header, stating which version of the SRTLink OS was used to generate the log file and the name of the model embedded in the OS. The sample interval and the report interval refer to the sample time of the model and the real-time report interval respectively.

The LogFileConfig option was set to "Yes" for this simulation and the current settings of the simulator can be seen following the header. Following that is the starting time and date for the log and the actual logged report intervals. These each equate to one second of real time, with the time in milliseconds giving the amount of each second spent performing the model simulation. The percentage given on each line is the amount of "sleeping" time for each report interval.

```
Sleeping Real-Time Simulator Version 3.01
Model: SimEvap                                                Header
Sample interval (ms): 100
Report interval (ms): 1000

Logfile Name:                 logfile.txt
Logfile active:               Yes
Clear file before logging:    No
Write config to file:         Yes
Logfile Mode:                 All events                      Simulator
Console Mode:                 All events                      settings
Write Config to Console:      Yes
Pause before simulating:      No
Report interval (steps):      10
Log started 13:12:19, 04/11/99
Simulating: 380ms,      Sleeping: 62.0%
Simulating: 40ms,       Sleeping: 96.0%
** Simulating: 1090ms, Overrun: 90 (9.0%) at 13:12:21          Simulation
Simulating: 40ms,       Sleeping: 96.0%                        loop results
Simulating: 50ms,       Sleeping: 95.0%
Simulating: 50ms,       Sleeping: 95.0%
Simulating: 41ms,       Sleeping: 95.9%
Simulating: 40ms,       Sleeping: 96.0%
Simulation finished 13:12:27, 04/11/99                         Footer
Error count for simulation: 1 of 8 steps
```

Figure 33 Example log file from the SRTLink OS

Note that there was an overflow in the third second of 90 milliseconds. This has been flagged in the log file and the actual time it occurred noted. Finally there is the footer giving the time that the simulation was ended and the total error count for the simulation.

APPENDIX D: SRTLINK TECHNICAL MANUAL

Table of Contents

General Notes

These notes are in no particular order, but could be useful for problem solving or technical background on the simulator. Most of the points listed here are also mention elsewhere within the thesis.

- Note that the sample times of the inputs to the driver blocks need to be set to a multiple of the sample time of the model.
- The node name can be defined differently for FromFix and ToFix. Although this is perfectly legal and both will continue to work independently of each other, there is no practical reason for doing this. Results of configuring the drivers like this are undefined, especially if there is a feedback to the model through the drivers.
- When running a model in Simulink with the drivers in it, the DLL's must be available in the Matlab path.
- Any non-EDA errors generated (ie. fatal errors occurring in *mdlStart*) are handled via *ssSetErrorStatus* and *ssSetStopRequested.* The error trap is in the execution loop of srt_main.c, and the error handler is the code following the loop. Any EDA errors that occur (ie. within *mdlOutputs*) are printed to the screen (not the log file), and operation continues. Errors at this point should only be range errors.
- Although it is possible to save all the VSP information for the EDA calls, hereby removing the need for calling eda_lookup, this step was seen as unnecessary. Because the tags do not change once initialised, one lookup is all that is needed,

and this is done in *mdlStart* before the actual simulation is run. During the actual simulation, there is no speed gain by pre-reading the VSP information from a file. Also, by leaving the tags to be read at the beginning of each simulation, any changes that are made to the control database are coped with without needing a currency check to be performed.

- Note that only FromFix is capable (at this point) of looking after ASCII *A_xx* fields – it was seen as unnecessary to implement this in ToFix.
- The EDA calls in the simulator seem to induce some clock skew – possibly because the EDA calls are initialised while the model is running, but are not actually performed (due to the multitasking operating system and the EDA call structure). Instead, they are run in separate threads while the model is sleeping, which skews the timing of the simulator thread. The sleep command refers only to the time in that thread, not absolute time and this being a serial processor performing time-slicing, clock skew occurs. The magnitude of this skew is small however – in the region of a millisecond a second.

FromFix

Purpose

S-function driver to be included in the Simulink model, to take values from Fix via EDA and make them available in the model.

Refer to the beginning of the code file "FromFix.c" for a revision history.

Function List

The performed tasks are broken into the routines under which they are performed, for the sake of clarity. Note that each of these function names refer to the entrypoints in the CMEX file. Refer to Appendix A for the code listing.

mdlInitalizeSizes

- Allocates one output port for the values received from Fix32, and one input port for control purposes.
- All of the port widths are set to DYNAMICALLY_SIZED. This allows the number of muxed channels being input into the block to set the size of the port subject to the maximum set with the NTF_LIST_LENGTH #define.

mdlInitalizeSampleTimes

The sample time of the block is set to the sample time of the model which is found with the *ssGetSampleTime(S)* routine. Doing this allows for the block to be triggered on every time step if so configured by the driver control panel (triggering a block faster than the fixed-step sample time of the model is not allowed by Simulink).

mdlSetWorkWidths

Set the size of RWork, IWork and PWork vectors. The PWork vector is used to pass variables from *mdlStart* to *mdlOutputs*, once they have been set with the parameters read in from the configuration file.

Length of the PWork vector is set to 9. The indexes within the vector refer to:

1. GroupHandle
2. NTFHandle
3. NodeName
4. TagList
5. FieldList
6. NTFCount
7. PackedNTFList
8. ValidNTFList
9. VNTFCount

mdlStart

This routine is used to set up the driver, based on the data stored in the configuration file. Once configured, pointers in the PWork vector are pointed at the data to make it available to *mdlOutputs*.

- Reads list of tags and fields in from a file (by default from "FromFix.cfg"), and stores them in *TagList* and *FieldList*. The name of the node to be used for all the EDA calls is also read in. As these lists are created, the *PackedNTFList* is also built. Range and error checking is done as the tags are read in (note this does not include EDA error checks). Reads from the file are in the form "1 = tag.field". The number that is associated with a tag refers to the muxed line that the value for this tag is output from the driver.
- Defines a group *GroupHandle*, then defines NTF's and adds the NTF handles to the group via *NTFHandle*. *A* pointer to the group is then inserted into *PWork* so it can be accessed in *mdlOutputs*.
- Sets the access rights for the database. This is not strictly necessary for versions of Fix beyond 3.x, but was included for the sake of completeness.
- Does an *eda_lookup* of the group, so that the VSP's and PDBSN are known and stored before speed becomes a constraint.
- Loops through the *PackedNTFList* array checking all tags for EDA errors. This generates an array called *ValidNTFList*, which contains all the tags checked as valid by Fix32.
- Note that *TagIndex* starts at 1 and goes up from there. This refers to which input into the mux block in the driver that that tag/field is for. Number 1 is the first, 2 the second, and so on. However, inside the S-function these values must have one subtracted from them, because arrays in C/C++ start at 0. So the first tag is stored in the arrays at index 0, the second at index 1, etc.
- Pointers to all the variables are put into *PWork* so that they can be pulled out by any of the other routines in FromFix.

mdlOutputs

- Using the pointers in PWork, setup data is recovered as defined in *mdlStart*. Although not strictly necessary, for the sake of readability the variables are put into pointers of the same name as in *mdlStart*.
- The GroupHandle (ie. the values relating to the list of tags in GroupHandle) is read from Fix32 and the individual tags are taken from the buffer and put onto the output port of the block. *eda_get_ascii* and *eda_get_float* are used depending on the prefix of the name of the field concerned.

mdlTerminate

- The GroupHandle pointer is pulled out of the PWork vector, and checked for NULL – if it's not NULL (ie. a valid handle), then it's deallocated.

Defines

Defines are listed in the order that they are defined within the code.

CONFIG_FILE_NAME	"fromfix.cfg"	
NUM_TAGS_OUT	ssGetOutputPortWidth(S,0)	// width of the mux feed into the driver
TF_SEP	"."	// seperator for tag/field in the config file
DEF_NODE_NAME	"MODEL"	// default node name
NUM_PARAMS	(0)	// number of parameters expected from the block
NTF_LIST_LENGTH	(300)	// max number of NTF's
NTF_STRING_SIZE	(40)	// max length of NTF strings
GH_IDX	(0)	// index in the PWork vector for passing GroupHandle
NTF_IDX	(1)	// index in the PWork vector for passing NTFHandle
NN_IDX	(2)	// index in the PWork vector for passing NodeName
TL_IDX	(3)	// index in the PWork vector for passing TagList
FL_IDX	(4)	// index in the PWork vector for passing FieldList
NTFC_IDX	(5)	// index in the PWork vector for passing NTFCount
PNTFL_IDX	(6)	// index in the PWork vector for passing PackedNTFList
VNTFL_IDX	(7)	// index in the PWork vector for passing ValidNTFList

VNTFC_IDX	(8) VNTFCount	// index in the PWork vector for passing
PWORK_SIZE	(9)	
RWORK_SIZE	(0)	
IWORK_SIZE	(0)	
CONTROL_PORT	(0)	// defines which input port is the control port
CP_SIZE	(1)	// width of control port
CP_ENABLE	(0)	// index in ControlPort for enable toggle
BUFFER_LENGTH	(40)	// length of ASCII I/O buffer to read values

Notes

Because a recompile is needed if any of the model settings or the S-function code is changed it was seen as pointless to put the name of the configuration file for the driver as an option in the model, given that all the other options are set in the S-function code. So there are no options that are set in the model, aside from the sample interval – all are set as defines in the code itself.

It was decided to implement float/ASCII readability for the FromFix driver because some of the fields in a block (the EGU limits for instance) are not stored as floats, but only as ASCII. This was not done for ToFix because model responses are all sent to AI or DI input blocks, which use float values only.

This driver is based on ToFix.c, so most of the code (especially in *mdlStart*) is identical to that in ToFix.c. The only major difference is in *mdlOutputs*, where data is read rather than written with EDA. Other minor differences are in defining the number of input and output ports and so forth.

ToFix

Purpose

Driver S-function to be included in the Simulink model, to take values from the model and send them to Fix32 via EDA.

Refer to the beginning of the code file "ToFix.c" for a revision history.

Function List

The performed tasks are broken into the routines under which they are performed, for the sake of clarity. Note that each of these function names refer to the entrypoints in the CMEX file. Refer to Appendix A for the code listing.

mdlInitalizeSizes

- Allocates two input ports, one for the values to be sent to Fix32 and one for control purposes.
- All of the port widths are set to DYNAMICALLY_SIZED. This allows the number of muxed channels being input into the block to set the size of the port subject to the maximum set with the NTF_LIST_LENGTH #define.

mdlInitalizeSampleTimes

The sample time of the block is set to the sample time of the model which is found with the *ssGetSampleTime(S)* routine. Doing this allows for the block to be triggered on every time step if so configured by the driver control panel (triggering a block faster than the fixed-step sample time of the model is not allowed by Simulink).

mdlSetWorkWidths

Set the size of RWork, IWork and PWork vectors. The PWork vector is used to pass variables from *mdlStart* to *mdlOutputs*, once they have been set with the parameters read in from the configuration file.

Length of the PWork vector is set to 9. The indexes within the vector refer to:

1. GroupHandle
2. NTFHandle
3. NodeName
4. TagList

5. FieldList
6. NTFCount
7. PackedNTFList
8. ValidNTFList
9. VNTFCount

mdlStart

This routine is used to set up the driver, based on the data stored in the configuration file. Once configured, pointers in the *PWork* vector are pointed at the data to make it available to *mdlOutputs*.

- Reads list of tags and fields in from a file (by default from "ToFix.cfg"), and stores them in *TagList* and *FieldList*. The name of the node to be used for all the EDA calls is also read in. As these lists are created, the *PackedNTFList* is also built. Range and error checking is done as the tags are read in (note this does not include EDA error checks). Reads from the file are in the form "1 = tag.field". The number that is associated with a tag refers to the muxed line that the value for this tag is taken from.
- Defines a group *GroupHandle*, then defines NTF's and adds the NTF handles to the group via *NTFHandle*. *A* pointer to the group is then inserted into *PWork* so it can be accessed in *mdlOutputs*.
- Sets the access rights for the database. This is not strictly necessary for versions of Fix beyond 3.x, but was included for the sake of completeness.
- Does an *eda_lookup* of the group, so that the VSP's and PDBSN are known and stored before speed becomes a constraint.
- Loops through the *PackedNTFList* array checking all tags for EDA errors. This generates an array called *ValidNTFList*, which contains all the tags checked as valid by Fix32.
- Note that *TagIndex* starts at 1 and goes up from there. This refers to which input into the mux block in the driver that that tag/field is for. Number 1 is the first, 2 the second, and so on. However, inside the S-function these values must have one

subtracted from them, because arrays in C/C++ start at 0. So the first tag is stored in the arrays at index 0, the second at index 1, etc.

- Pointers to all the variables are put into *PWork* so that they can be pulled out by any of the other routines in FromFix (in this version this only relates to *mdlOutputs*).

mdlOutputs

- Using the pointers in PWork, setup data is recovered as defined in *mdlStart*. Although not strictly necessary, for the sake of readability the variables are put into pointers of the same name as in *mdlStart*.
- Values are read from the input port at each of the defined and valid tag indexes on the port, and put into *GroupHandle* ready to be communicated to Fix. *eda_write* is called to communicate the values and all the tags are checked for errors.

mdlTerminate

- The GroupHandle pointer is pulled out of the PWork vector, and checked for NULL – if it's not NULL (ie. a valid handle), then it's deallocated.

Defines

Defines are listed in the order that they are defined within the code.

CONFIG_FILE_NAME	"tofix.cfg"	
NUM_TAGS_IN	ssGetInputPortWidth(S,0)	// width of the mux feed into the driver
TF_SEP	"."	// seperator for tag/field in the config file
DEF_NODE_NAME	"MODEL"	// default node name
NUM_PARAMS	(0)	// number of parameters expected from the block
NTF_LIST_LENGTH	(300)	// max number of NTF's
NTF_STRING_SIZE	(40)	// max length of NTF strings
GH_IDX	(0)	// index in the PWork vector for passing GroupHandle
NTF_IDX	(1)	// index in the PWork vector for passing NTFHandle
NN_IDX	(2)	// index in the PWork vector for passing NodeName
TL_IDX	(3)	// index in the PWork vector for passing TagList

FL_IDX	(4)	// index in the PWork vector for passing FieldList
NTFC_IDX	(5)	// index in the PWork vector for passing NTFCount
PNTFL_IDX	(6)	// index in the PWork vector for passing PackedNTFList
VNTFL_IDX	(7)	// index in the PWork vector for passing ValidNTFList
VNTFC_IDX	(8)	// index in the PWork vector for passing VNTFCount
PWORK_SIZE	(9)	// size of PWork Vector (1+largest index value above)
RWORK_SIZE	(0)	// size of RWork Vector
IWORK_SIZE	(0)	// size of IWork Vector
DATA_PORT	(0)	// defines which input port is for data
CONTROL_PORT	(1)	// defines which input port is the control port
CP_WIDTH	(1)	// width of control port
CP_ENABLE	(0)	// index in ControlPort for enable toggle

Notes

Because a recompile is needed if any of the model settings or the S-function code is changed, it was seen as pointless to put the name of the configuration file for the driver as an option in the model, given that all the other options are set in the S-function code. So there are no options that are set in the model, aside from the sample interval – all are set as defines in the code itself.

Important: The value that is passed to Fix, via *eda_set_float*, must be in a float and passed as "&floatvar" (see code fragment below, in the last line of the for loop). Casting the variable does not seem to work, nor does any other data type (int, double, etc). Therefore a variable of type float is defined and before sending a value (of any type) to Fix32, it is put into this variable.

```
static void mdlOutputs(SimStruct *S, int_T tid) {
      float       Value;
.
.
      for (i=0;i<*VNTFCount;i++) {
            NTFIndex=ValidNTFList[i];
            Value=(float)*uPtrs[NTFIndex];
            err=eda_set_float(GroupHandle,NTFHandle[NTFIndex],
                                                              &Value);
      }
.
.
.
```

Note that the type GNUM is a pointer to a structure, not the actual EDA group structure. Because of this references to it in *mdlOutputs* do not need the * operator, as the other single value variables passed through *PWork* do. Note also that arrays do not need the * operator as the name of an array is a pointer to the first element in the array – references into arrays are therefore pointer arithmetic. The *uPtrs* type (defined in "simstruct.h") is a pointer to an array of pointers. It therefore must be used in the form x = *uPtrs[0] to reduce the indirection by one level.

For an unknown reason, "while (!_kbhit()) {}" doesn't work in the S-function error handler to insert a pause for a keypress. Therefore, rather than just letting the window close without giving the user to see any errors displayed, a 5 second pause has been inserted.

Note that the ToFix driver cannot communicate anything except float values to Fix. This is not viewed as a problem or restriction because the Simulink model does not produce anything except floats, and Fix32 does not expect input into non-float tag/fields. However, it is possible to specify a non-float tag/field in the config file and the driver will try to output a value to it. This will at most result in a consistently generated "wrong data type" error from Fix.

InitFix

Purpose

Initialise Fix32 to start with values for the simulation (typically these are PID controller parameters and setpoints).

Refer to the beginning of the code file "InitFix.c" for a revision history.

Function List

The performed tasks are broken into the routines under which they are performed, for the sake of clarity. Note that each of these function names refer to the entrypoints in the CMEX file. Refer to Appendix A for the code listing.

mdlInitalizeSizes

- Allocates one output port for the values received from Fix32, and one input port for control purposes.
- All of the port widths are set to DYNAMICALLY_SIZED. This allows the number of muxed channels being input into the block to set the size of the port subject to the maximum set with the NTF_LIST_LENGTH #define.

mdlInitalizeSampleTimes

The sample time of the block is set to the sample time of the model which is found with the *ssGetSampleTime(S)* routine. Doing this allows for the block to be triggered on every time step if so configured by the driver control panel (triggering a block faster than the fixed-step sample time of the model is not allowed by Simulink).

mdlStart

This routine is used to set up the driver, based on the data stored in the configuration file. Unlike FromFix and ToFix however, there is no communication from *mdlStart* to *mdlOutputs.* All of the tasks performed by InitFix are done in *mdlStart*; *mdlOutputs* is empty. The *PWork* vector is therefore not used.

- Reads list of tags and fields in from a file (by default from "ToFix.cfg"), and stores them in *TagList* and *FieldList*. The name of the node to be used for all the EDA calls is also read in. As these lists are created, the *PackedNTFList* is also built. Range and error checking is done as the tags are read in (note this does not include EDA error checks). Reads from the file are in the form "1 = tag.field,x". The number that is associated with a tag refers to the muxed line that the value for this tag is taken from. The initialisation value (x) associated with each tag is stored in the *OnceValue* array.

- Defines a group *GroupHandle*, then defines NTF's and adds the NTF handles to the group via *NTFHandle*
- Sets the access rights for the database. This is not strictly necessary for versions of Fix beyond 3.x, but was included for the sake of completeness.
- Does an *eda_lookup* of the group, so that the VSP's and PDBSN are known.
- Loops through the *PackedNTFList* array checking all tags for EDA errors. This generates an array called *ValidNTFList*, which contains all the tags checked as valid by Fix32.
- Note that *TagIndex* starts at 1 and goes up from there. This refers to which input into the mux block in the driver that that tag/field is for. Number 1 is the first, 2 the second, and so on. However, inside the S-function these values must have one subtracted from them, because arrays in C/C++ start at 0. So the first tag is stored in the arrays at index 0, the second at index 1, etc.
- Values are taken from the *OnceValue* and inserted into *GroupHandle*. The EDA group is then written out to Fix32, and the write checked for errors. Any errors that have occurred are displayed in the console window. *GroupHandle* is then released.

mdlOutputs

This function is does not contain any code because all of the initialisation has already occurred in *mdlStart*. Note that it is defined, however; Simulink will not use a CMEX file if it does not have this entrypoint.

mdlTerminate

This function is does not contain any code because no variables have been inserted into *PWork*. Note that it is defined, however; Simulink will not use a CMEX file if it does not have this entrypoint.

Defines

Defines are listed in the order that they are defined within the code.

CONFIG_FILE_NAME	"initfix.cfg"	
NUM_TAGS_IN	ssGetInputPortWidth(S,0)	// width of the mux feed into the driver
TF_SEP	"."	// seperator for tag/field in the config file
DEF_NODE_NAME	"MODEL"	// default node name
NUM_PARAMS	(0)	// number of parameters expected from the block
NTF_LIST_LENGTH	(300)	// max number of NTF's
NTF_STRING_SIZE	(40)	// max length of NTF strings
PWORK_SIZE	(0)	// size of PWork Vector (1+largest index value above)
RWORK_SIZE	(0)	// size of RWork Vector
IWORK_SIZE	(0)	// size of IWork Vector
CONTROL_PORT	(0)	// defines which input port is the control port
CP_WIDTH	(1)	// width of control port
CP_ENABLE	(0)	// index in ControlPort for enable toggle
ONCE_PORT	(0)	// number of output port used for writeonce list

Notes

This driver is a cut-down version of ToFix. The modifications are:

1. *mdlOutputs* does not contain any code. During the simulation itself, when *mdlOutputs* is called by Simulink there are no tasks for InitFix to perform; all the tasks are performed before the simulation loop is entered.

2. While reading in the tag list, the initialisation values are also read in parallel. They are stored in the *OnceValue* array to be written to Fix32 after closing the configuration file.

3. Unlike FromFix and ToFix, which do not perform any EDA reads or writes in *mdlStart*, InitFix writes the values stored in *OnceValue* to Fix32 at the end of the routine.

Although there is no code in both *mdlOutputs* and *mdlTerminate*, both are still defined in the source code. This is because Simulink will not use a CMEX file if it does not have the required entrypoints defined in it, regardless of whether they perform any functions or not.

Srt_main

Purpose

Real time kernel of the linked application, used to handle errors and timing in the model at runtime.

Refer to the beginning of the code file "srt_main.c" for a revision history.

Function List

- Looks in the current directory for a configuration file (by default called "SRT.cfg"). If the file is found, then the settings are read from it and used to configure the simulator. If no configuration file is found, then the user is notified and a new standard configuration file written, using the default settings. Any parameters that are not explicitly set in the configuration file or have invalid settings default to set values (valid settings and defaults listed below). If the file cannot be found, then a new default one is created. The parameters that are set are:
    - LogFileName: Name of the log file (valid filename, default "logfile.txt")
    - LogFile: Whether or not to write events to a log file ("Yes" or "No", default "Yes")
    - LogFileClear: Clear the log file at the beginning of each simulation ("Yes" or "No", default "No")
    - LogFileConfig: Write simulator configuration details to the log file at the beginning of each simulation ("Yes" or "No", default "Yes")

- LogFileMode: Which class of events to write to the log file ("Errors" or "All", default "All")

- ConsoleMode: Which class of events to write to the console window ("Errors" or "All", default "All")

- ConsoleConfig: Write configuration details to console window at the beginning of a simulation ("Yes" or "No", default "Yes")

- SimStartPause: Wait for user to press a key before beginning the simulation ("Yes" or "No", default "Yes")

- SimReportStep: Sets the interval (number of steps) between writing to the log file (positive integer, default 10)

- Runs the model in a loop. The model is run for SimReportStep intervals to find the values for the current time step and the time taken to compute this information measured. Dependant on the configuration settings, timing information for that step is written to a log file and to the console window. The current time is then taken again, to take into account the time taken to execute the overhead code and then a delay is calculated to ensure the next series of steps begin calculation at the correct point in real-time to synchronise the simulation with real-time.

Upon exiting the loop, memory is cleaned up, the log file (if any) has the finish time of the simulation written to it, and is closed.

Defines

Defines are listed in the order that they are defined within the code.

```
EA_ERRORS          (0)                       // action only on errors
EA_ALL             (1)                       // action on all steps
NO                 (0)                       // used for config stuff
YES                (1)                       // ditto
LF_NAME_LENGTH     (40)                  // length of the log file name
struct CFG_STRUCT {
                   char LogFileName[LF_NAME_LENGTH];     // name of the log
                   file
```

```
                        int LogFile;            // Y/N flag: write a log file or not
                        int LogFileClear;       // Y/N flag: clear log file before writing
                        int LogFileConfig;      // Y/N flag: write configuration to log file
                        int LogFileMode;        // EA setting
                        int ConsoleMode;        // EA setting
                        int ConsoleConfig;      // Y/N flag: display config settings at start
                        int SimStartPause;      // Y/N flag: wait before beginning
                        int SimReportStep;      // positive integer for report interval
} Cfg = {"logfile.txt",YES,NO,YES,EA_ALL,EA_ALL,YES,YES,10}; // defaults - if not set in
                                                              CONFIG_FILE, left as is
CONFIG_FILE          "srt.cfg"                  // name of config file
```

Notes

- Because the simulation is only ever going to be run with a fixed step solver and there are no variable time blocks in the model, it was pointless to tie anything to an interrupt, although the code would have been somewhat more flexible. For this reason, the file that was used as a base for the development of srt_main.c, was grt_main.c (Generic Real-time). Alterations that were made are listed in the revision history in the file itself.

- The options set in the configuration file were put in a file rather than in options entered into a block mask in Simulink so that after the model has been compiled, it does not need to be re-compiled to change the options. The structure of the configuration file reading code is easily extensible for further development in this direction.

- The code for the configuration file is considerably more complex than strictly necessary, but this was in order to make it as robust as possible from the viewpoint of the operators. The addition of comments (and the attendant parsing code) was considered useful, again from the viewpoint of the operators.

- The time taken for the model to execute is taken directly after the execution of the model for one time interval (SimReportStep multiplied by the sample time of the mode) and it is this value that is written to the log file. However, this value does not take into account the delay from the overhead code to write to the log file and do the timing calculations. Therefore after the overhead code has executed, the time is taken again and the sleeping delay is calculated with this. This is done to reduce clock skew on the simulation, although the overhead codes runs in well under 5ms.

- Most models compiled to run as a simulator will have fast system dynamics and so the fixed sample speed of the model must be set quite small to stop errors accumulating (~0.1 seconds). Reporting to the display and to a log file at this frequency was seen as overkill, so the provision for SimReportStep was included. Note that the interval for communication to Fix32 (set in the Driver Control Panel in the model) is independent of the SimReportStep interval.

- Errors from the S-function drivers are reported to "Srt_main.c" via the *ssSetErrorStatus* and *ssSetStopRequested*. The error handlers follow the main execution loop, and handle the generated error. Nothing is done with in the S-function drivers themselves.

- The error handler that is evoked at the end of the simulation loop dependent on the exit condition (ssSetStopRequested, ssSetErrorStatus, Ctrl X) does not seem to correctly run code in front of the exit(1) statements in each handler. This must be a compiler or linker issue, but I can find no solution. Hence the use of Sleep to cause a delay on an errored exit – an API call is not run through the compiler.

- The Ctrl-X check is at the end of the simulation loop *after* the sleeping period to allow the user a time interval over which they can press Ctrl-X. In general, the delay will be longer than the model so it makes sense to put it after the delay rather than after the mode. The drawback to this approach is that if the user pushes Ctrl-X while the model is executing, the delay for that step will still occur before the simulation loop is exited the next time around.

APPENDIX E : FIX32 DATABASE AND CONTROL SCREEN MODIFICATIONS

Fix32 Database Modifications

These modification were made to the POWDER database, dated 15/10/98. For information on how to make the modifications listed here, refer to the *Database Builder Manual* supplied with Fix32.

Note: The new blocks should be added to the database *before* the global modifications are made. This ensures that the numbering of the I/O addresses is consistent.

Blocks added

These blocks have no bearing on the control of the model; they are used for the control of the model, via the SimEvap Controls page added to the Fix32 control system. Each of the blocks is an AR (Analog Register) block, and has the tag name of "MC_xxx" to differentiate them from the existing blocks.

- MC_HOLD Used to store the holding tube time (seconds).
- MC_VV1 Used to store the nominal value of the steam pressure into the DSI (bar).
- MC_VP1 Percentage variation of the DSI steam pressure. This is a percentage of the nominal value.
- MC_VR1 Rate of variation of the DSI steam pressure (seconds). This is the smallest amount of time in which the variable can change from its minimum variation value to it's maximum variation.
- MC_VV2 Used to store the nominal value of the steam pressure to the TVR (bar).

- MC_VP2 Percentage variation of the TVR steam pressure. This is a percentage of the nominal value.
- MC_VR2 Rate of variation of the TVR steam pressure (seconds). This is the smallest amount of time in which the variable can change from its minimum variation value to it's maximum variation.
- MC_VV3 Used to store the nominal value of the water temperature from the Cogeneration plant (°C).
- MC_VP3 Percentage variation of the Cogen water supply temperature. This is a percentage of the nominal value.
- MC_VR3 Rate of variation of the Cogen water supply (seconds). This is the smallest amount of time in which the variable can change from its minimum variation value to it's maximum variation.
- MC_VV4 Used to store the nominal value of the product flow from the concentrate tank into the dryer (litres/hour).
- MC_VP4 Percentage variation of the product flow to the dryer. This is a percentage of the nominal value.
- MC_VR4 Rate of variation of the product flow (seconds). This is the smallest amount of time in which the variable can change from its minimum variation value to it's maximum variation.
- MC_VV5 Used to store the nominal value of the flow of milk from the silos into the feed tank (litres/hour).
- MC_VP5 Percentage variation of the milk flow from the silos. This is a percentage of the nominal value.
- MC_VR5 Rate of variation of the milk flow from the silos (seconds). This is the smallest amount of time in which the variable can change from its minimum variation value to it's maximum variation.
- MC_VV6 Used to store the nominal value of the dry matter in the milk flow from the silos (percentage solids).
- MC_VP6 Percentage variation of the silo flow dry matter. This is a percentage of the nominal value.

- MC_VR6 Rate of variation of the silo flow dry matter (seconds). This is the smallest amount of time in which the variable can change from its minimum variation value to it's maximum variation.

One other block was added of type AI, called DUMMYF. This block was used simply as a non-functional dummy block for the fault tags relating to other databases.

Modifications made to existing blocks

All of these changes were global to the database. To ensure this the Query setting must be {Tag Name} = "*"

1. The Init. Auto/Manl field of all of the blocks in the database were set to "MANL", as opposed to "AUTO". This enables EDA to write values into the database (automatic mode blocks will only accept input from the SAC task).

2. All instances of "E" in the Scantime field were be altered to "1". If the Scantime of the block is "E" then the chain associated with the block will only trigger on an exception. The SAC can generate exceptions, however EDA cannot. Removing any occurrences of "E" allows a chain to trigger dependant on the value in the block.

3. The I/O Driver field was changed to SIM for every block in the database. However, the Database Builder will not allow a search and replace to be performed on the I/O Driver field of the database, nor will it allow a global replace. Therefore to make this change, the database was exported as text and the change made with a search and replace in the Windows Notepad.

4. Because the SIM driver does not use the same I/O Address format as the PLC drivers, the I/O address of each block in the database must be renumbered. Rather than do this manually (there are after all over 2000 blocks in the database) a utility called "NumberSim.exe" can be found on the SimEvap installation CD in the Tools directory. When this file is run, it requests an input file and an output file. Given the name of the text file export by the Database Builder, *NumberSim* will renumber the I/O addresses of the blocks in the database.

5. The modified text filed can then be imported into the Database Builder and saved as the new PDB file.

Control Screen Modifications

The changes listed here relate only to the control screens concerning the Powder 3 plant. These flies are:

- BP3menu.odf
- Smain.odf
- meaeff1.odf
- meacond.odf
- meaOverV.odf
- meafeed.odf
- meatvr.odf
- meaeff23.odf

For information on how to make the modifications listed here, refer to the *Draw Manual* supplied with Fix32.

Global changes

These changes were made with a Multipicture Search and Replace in Draw with respect to all of the ODF files listed above. Note that the quotes used in the following list are only used to delineate the text here; they are not actually part of the search and replace text. The searches were:

1. Find "POWDER:*", replace with "MODEL:*"
2. Find "P4A:*", replace with "MODEL:*"
3. Find "PSILO:*", replace with "MODEL:*"

Unfortunately, the search and replace performed by Draw does not check text strings within Dynamos. This means that all of the pumps and valves on each of the control screen must be manually altered. Each pump and valve has two strings in it in the form "POWDER:*" which must be altered to "MODEL:*".

Changes made to individual screens

BP3menu.odf

This is the main selection bar, normally displayed along the top of the screen.

- Colour of the text on all of the non-active buttons (ie. Powder 4, Powder 3 Dryer, Powder 5) turned to light grey.
- OnUp action field in the button backgrounds needs to be deselected to disable the button action.
- Addition of the "SimEvap Controls" button at the right hand end of the bar, in red text. The OnUp action field of the background needs to be made active and linked to the "SimCtrl.ODF" file.

Smain.odf

This is the picture file for the pulldown menu shown when the user clicks on the "Main" button on the main selection bar.

- Powder 4, Powder 5 Silos, Powder Transport, Other factories and Talk are all greyed out. This should be done with the same grey used for BP3menu.odf
- OnUp action field in the background of each selection should be deselected to disable the functionality.

Meaeff1.odf

This is the display screen relating to the MVR section.

- The level sensors for the liquid level in the shell and the condensate water sensor removed (P3LS1310, P3LS1311, P3LS1410, P3LS1411, P3LS1348).

- P3TE1381F and P3TE1382F are replaced with DUMMYF. These tags are found in the Dynamos for P3TT1381 and P3TT1382.

Meacond.odf

This is the display screen relating to the condenser in the MVR.

- Removed the CTW pump indicator.
- P3PT1782F and P3CT1791F are replaced with DUMMYF. These tags are found in the Dynamos for P3PT1782 and P3CT1791.

MeaOverV.odf

This is the Overview display screen.

- Removed the CTW indicator.
- The level sensors for the liquid level in the shell removed (P3LS1310, P3LS1311, P3LS1410, P3LS1411).
- Removed the "High" warning from the turbidity meter.
- Disabled the "Visible" logic for the "OK" on the turbidity meter.
- Changed the turbidity meter tag to DUMMYF.
- Altered the silo display Dynamos for both temperature and volume to DUMMYF.

Meafeed.odf

This is the display screen relating to the feed tank.

- Removed the "High" warning from the turbidity meter.
- Disabled the "Visible" logic for the "OK" on the turbidity meter.
- P3FT1184AF and P3CT1790F are replaced with DUMMYF. These tags are found in the Dynamos for P3FT1184 and P3CT1790.
- Changed the P3FT1787 and P3CT1183 Dynamos to operate from DUMMYF.

Meatvr.odf

This is the display screen relating to the preheating section.

- Removed the CTW indicator.
- P3TT1283F replaced with DUMMYF in Dynamo P3TT1283.
- P3TT1293F replaced with DUMMYF in Dynamo P3TT1293.
- P3TT1295F replaced with DUMMYF in Dynamo P3TT1295.
- P3TT1284F replaced with DUMMYF in Dynamo P3TT1284.
- P3TT1294F replaced with DUMMYF in Dynamo P3TT1294.
- P3TT1296F replaced with DUMMYF in Dynamo P3TT1296.

Meaeff23.odf

This is the display screen concerned with the TVR.

- P3TT1582F replaced with DUMMYF in Dynamo P3TT1582.
- P3TT1682F replaced with DUMMYF in Dynamo P3TT1682.
- P3FT11787F replaced with DUMMYF in Dynamo P3FT1787.