Titan Corporation - Houston Division

TRANSMITTAL MEMO

TM NO.	
DATE	**April 6, 2005**
CONTRACT NO.	
TASK ORDER NO.	

SUBJECT: **Trick Design Documentation**

ENCLOSURE: **Design Document**
Trick Simulation Environment
for the Trick 2005.0 Release

TO
NASA/LYNDON B. JOHNSON SPACE CENTER
2101 NASA ROAD 1
HOUSTON, TEXAS 77058
ATTN: Les Quiocho (ER7)

REMARKS:

This document details the "high level" software design of the Trick Simulation Environment including its executive features and utility processors. For a detailed design of Trick, refer to the commented code of Trick in the $TRICK_HOME/trick_source directories. For detailed syntax of how to use Trick, refer to the User's Guide.

Prepared by:

Keith Vetter
Software Engineer
Titan Systems Corporation

Eddie J. Paddock
Senior Principal Engineer
Titan Systems Corporation

This Page Intentionally Blank

The Trick
Design Document
Trick 2005.0 Release

Prepared by

Keith Vetter
Titan Corporation
1020 Bay Area Blvd. Suite 200
Houston, Texas 77058

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

JOHNSON SPACE CENTER

AUTOMATION, ROBOTICS and SIMULATION DIVISION

Delivered to NASA in Titan Transmittal Memo #######

This Page Intentionally Blank

Table of Contents

Section *Page*

1.0 Introduction

1.1 Scope

This document covers the concept and design of the Trick Simulation Environment. It offers a bird's eye view of the Trick design.

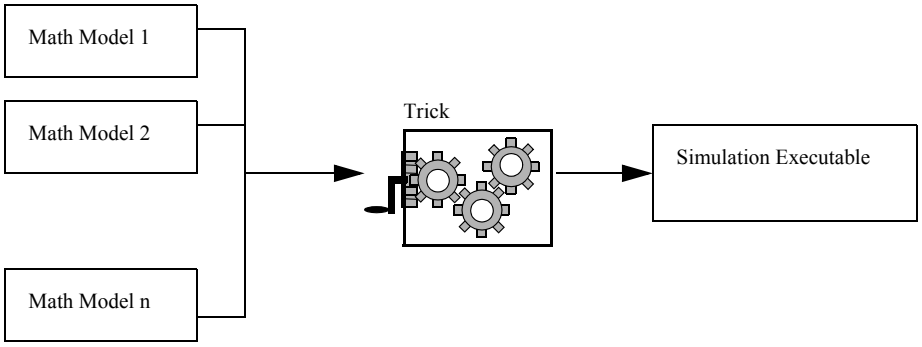
For details concerning items such as input variable syntax, source code syntax, simulation definition syntax etc., refer to the Users Guide. For details about how a particular design is implemented, refer to the code itself. For details on how to build a simulation, refer to the Tutorial. For details concerning the installation of Trick and what platforms are supported, refer to the installation portion of the User's Guide.

This is not a requirements document. In the past (pre-2001), requirements were part of the "Product Specification" Document. These have been archived, and are available by request. All new software requirements are stored in a database in the Trick lab. If you are interested in finding requirements or adding a new requirement, contact a Trick representative.

1.2 Concept

Trick was designed to allow simulation engineers to concentrate on the math model development rather than the simulation executive. Trick eliminates the simulation executive specific and runtime input/output (I/O) code development tasks from simulation development. It is designed to allow modelers to share their models between simulations.

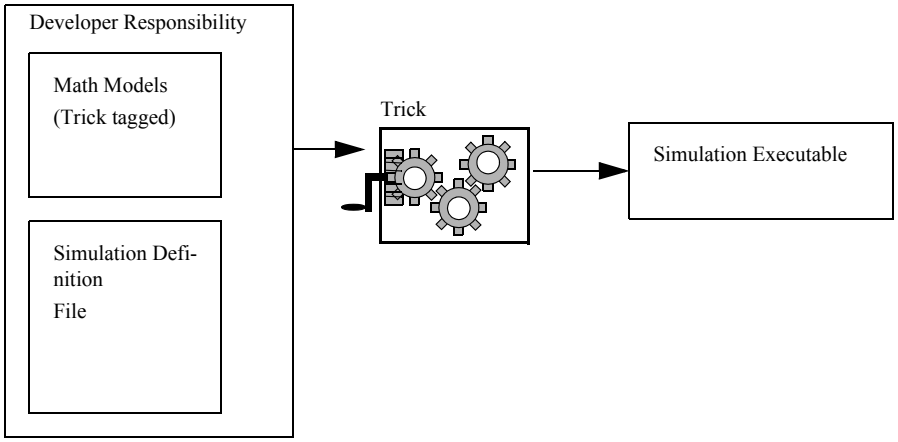
Figure 1 Concept



1.3 Developer/Trick Interface

To create a simulation, a developer will create models in C or FORTRAN 90 (F90 is only supported on SGI IRIX 6.5 OS, and will soon be unsupported). Model code is tagged with special instructions for Trick in the comment sections of the code. The developer will also create a text file called a simulation definition file. The simulation definition file is a blue print of how the developer plans to put the actual simulation together. The simulation definition file contains all sorts of specifications such as math model function (Trick uses the term "Job" to refer to C functions) calls, math model scheduling times, data structure declarations and default data. It houses everything that Trick needs to define a simulation with respect to its math model code.

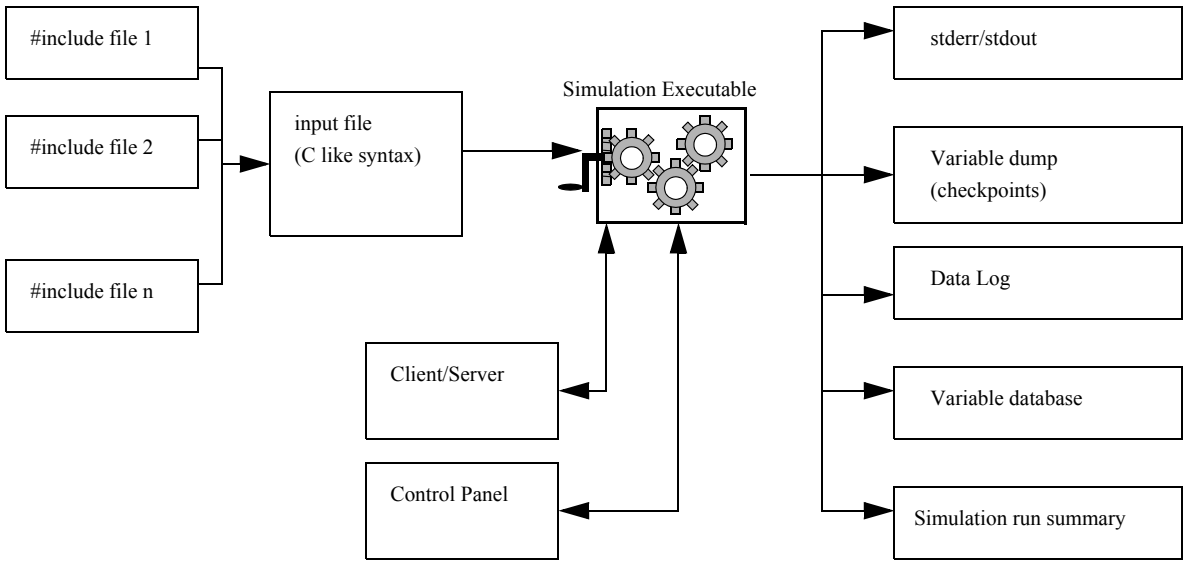
Figure 2 Developer/Trick Interface



1.4 Simulation Executable and Input/Output

The simulation executable is a binary executable created by a C compiler/loader. The simulation executable has one main calling argument which is a text file appropriately called an "input file". The input file's syntax closely resembles a C assignment statement and follows a simple "name = value;" format. It is able to #include other files which allows for a nested modular design of data files. The input file design and creation is the responsibility of the developer and/or user. The executable has several outputs. It prints to stderr, stdout. It is also able to send data across different communication mediums such as sockets. It is able to dump all the variables it has knowledge of into a text file coined a "checkpoint". Each time it runs, it dumps a text file that contains a summary about the particular simulation run. The executable may also dump a database of variable information. Finally, if requested, it may log data. While the simulation is running, a two-way communication is offered to the user through the Trick control panel. The executable has built in communication software as well that allows it to communicate to other processes in a client/server model.

Figure 3 Simulation Executable IO

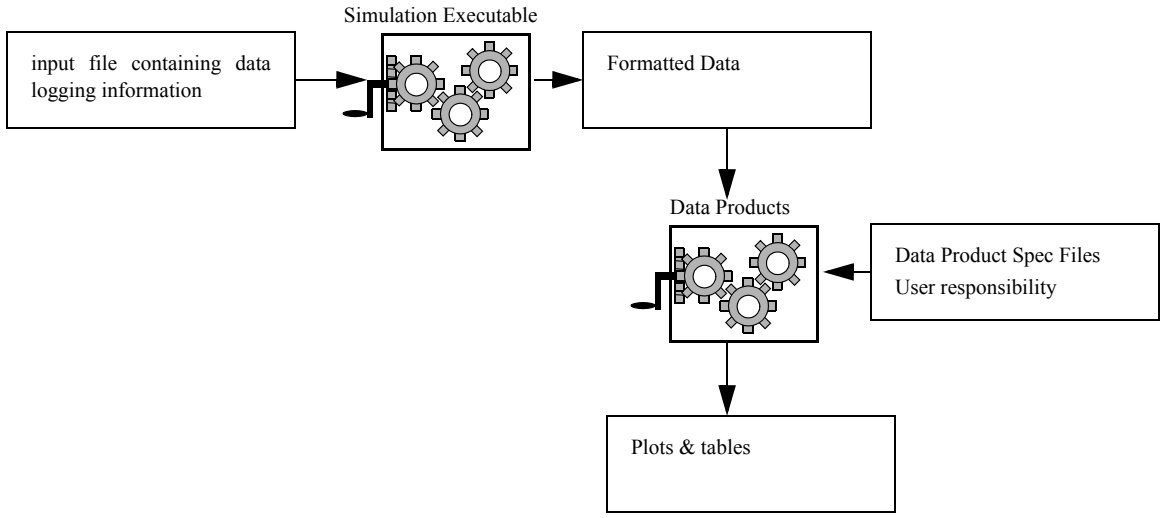


1.5 Developer/Data Interface

One of the outputs of the simulation executable is data. Users are most interested in model data resulting from model calculations. Internal simulation data may be logged as well. The choice of data logged, the format of data (e.g. fixed ascii, binary) and the frequency at which data is logged is specified by the user in the simulation input file (data driven).

Once data is logged, it may be viewed in the form of a plot or table. The Trick data_products program is responsible for processing data, creating plots etc. In order to view or post process data, it is necessary for the developer to create description (data product specification) files for the data_products program.

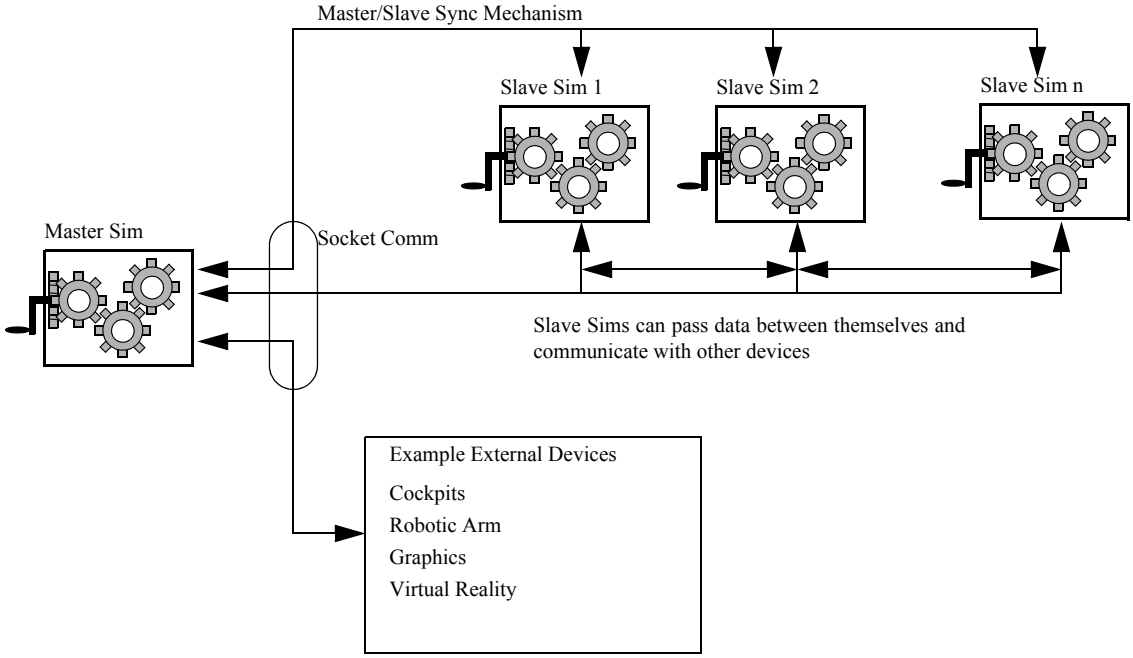
Figure 4 Developer/Data Interface



1.6 Communicating With External processes or devices

The simulation executable can be programmed to communicate with any external process or device using trick communications which is a socket based library. As an example, Trick simulations may be programmed to drive a cockpit. Special accommodations were made to allow Trick simulations to communicate with other Trick simulations. These distributed processing and synchronous/asynchronous capabilities of Trick will be discussed in more detail later, but Figure 5 below shows the high level concept.

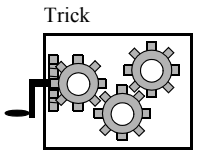
Figure 5 Simulation Executable And External Devices



2.0 Creating A Simulation Executable

This section will detail how Trick takes a set of models and creates a simulation executable. Referring to Figure 1, we will now find out what is in that Trick gear box. Popular opinion holds that it is black voodoo. Hopefully we can dispel at least some of that myth.

Figure 6 The Trick Box

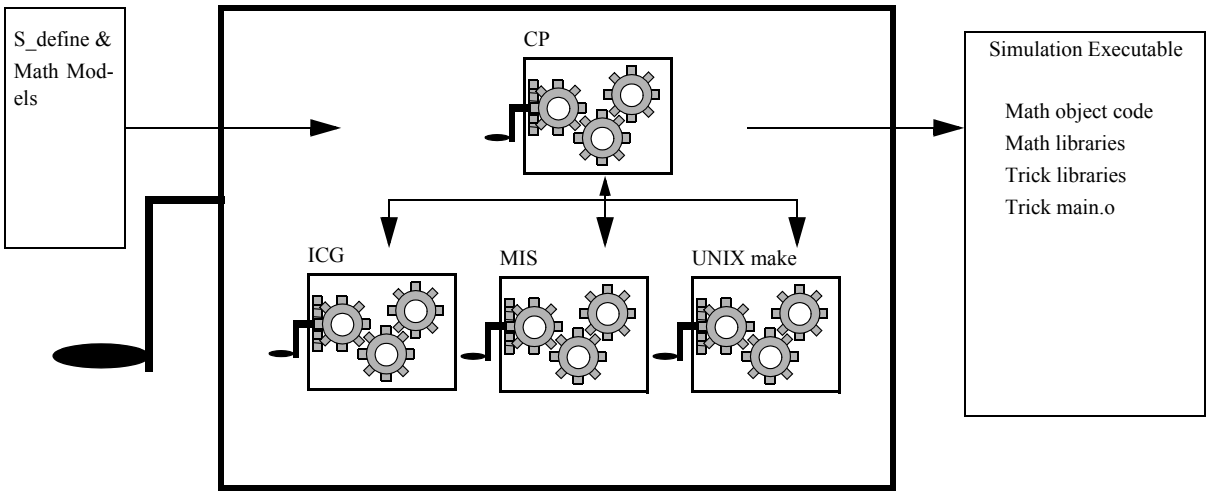


2.1 Trick Processor Overview

In a nutshell, Trick uses three code generating processors, UNIX make and the C compiler/loader to build the simulation executable. The three Trick processors are called the Interface Code Generator (ICG), Module Interface Specification (MIS) and the Configuration Processor (CP). All three processor are scripts written in Perl. ICG is used to parse header files and build the Trick internals for data structures. MIS is used to parse source code and build up the Trick internals for math model functions. CP parses the Simulation Definition file (S_define) and uses output from MIS and ICG along with the compiler/loader to create the simulation executable. The developer will call CP to create the simulation executable.

To understand these internals, a more detailed description of the simulation executable is needed. The simulation executable is an amalgam of math model and Trick executive object code. Trick must know all math model data structure information to be able to access variables from memory, therefore it uses ICG to auto-generate Input/Output or "IO" source code from each and every data structure header file in the math models. Trick must also know all function prototype information, therefore it uses MIS to build internals to be used for function calls. CP, the final processor, parses the simulation definition file, creates a special source file named S_source.c and then invokes the compiler/loader. In the end: math model object code/libraries; IO auto-generated object code; simulation math model object code; Trick executive libraries and a Trick main() are all linked together to form the simulation executable.

Figure 7 Inside The Trick Box



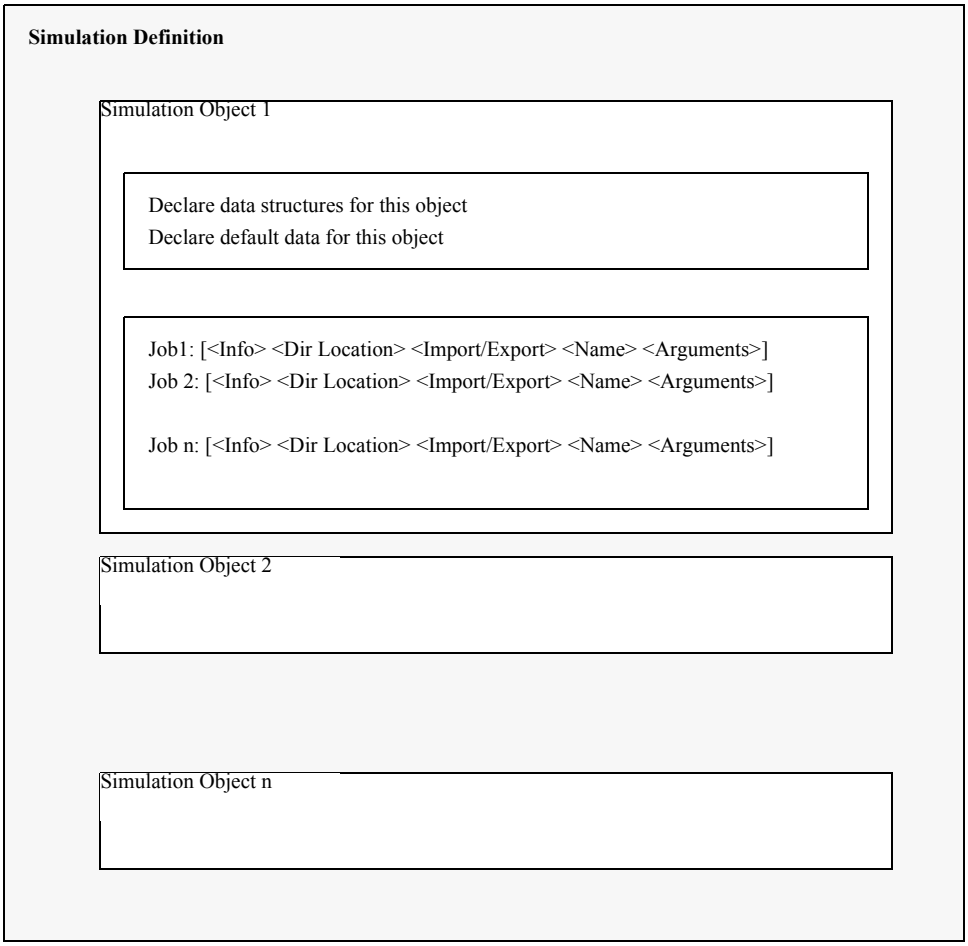
2.2 Configuration Processor

The Configuration Processor (CP) is responsible for parsing a simulation definition file. Using the simulation definition file as a blue print, it creates a simulation executable. CP also requires code and other information about the simulation data structures and math model modules (functions) which is generated by ICG and MIS. This code and database information will be discussed later.

2.2.1 Simulation Definition File (S_define)

This is the blue print that CP uses to create a simulation. For details concerning the syntax of the S_define, consult the User's Guide. An abstract view of the simulation definition is displayed below.

Figure 8 Simulation Definition (S_define)



As shown, the simulation definition is broken up into "simulation objects" which contain data structure, default data specifications and job (function or subroutine) information.

Simulation objects are a way of organizing jobs into cohesive units. The order of the objects are important. The Trick executive will call jobs first in "class" order as the primary sort, and then in S_define sequential order as the secondary sort. More about job classifications will be discussed later.

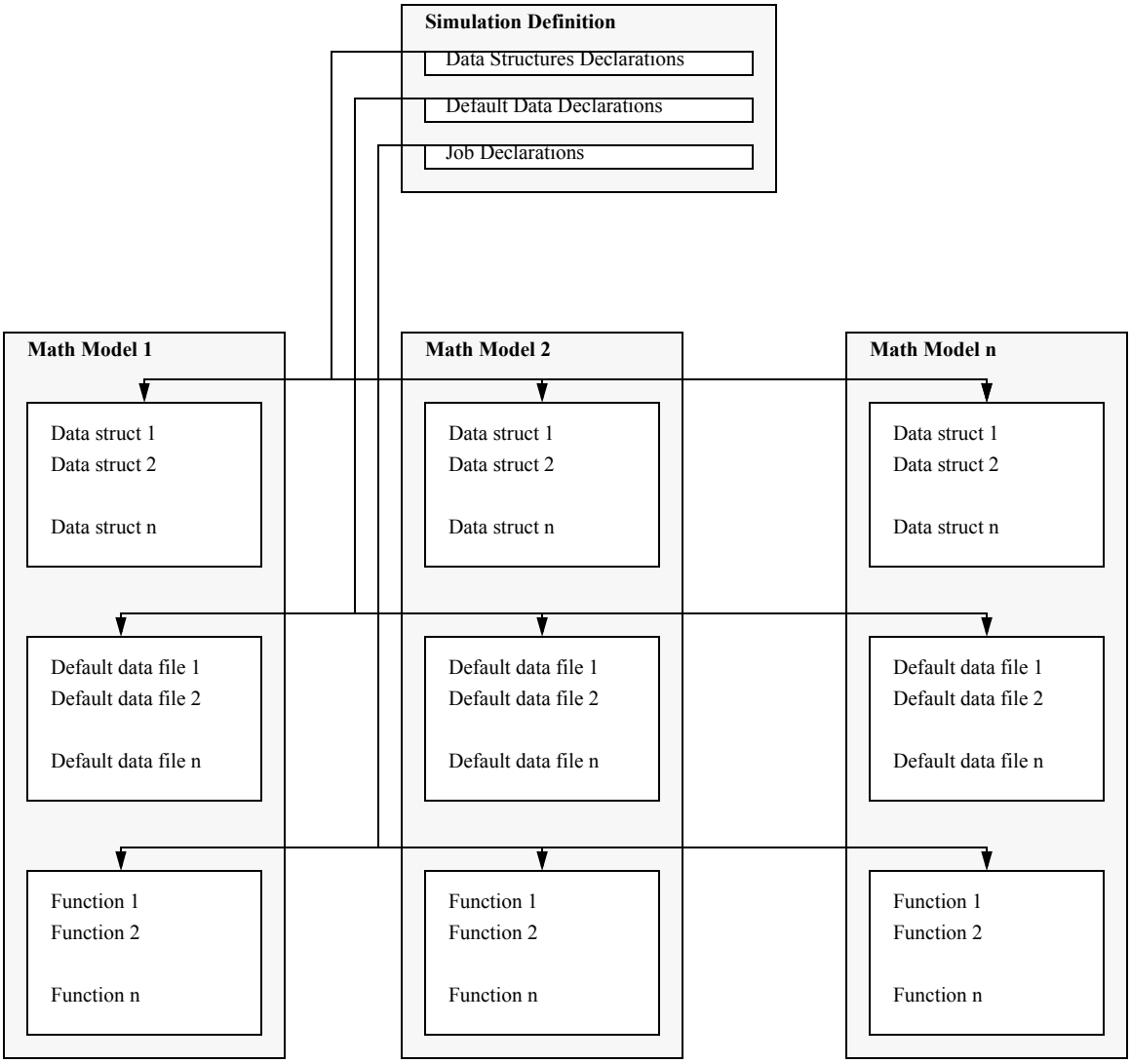
Data structures must be declared so that Trick knows which header files to instantiate for IO code and which header files to include.

Default data may be declared optionally so that Trick knows how to initialize data structures for simulation execution.

Jobs are the building blocks of the simulation objects. They point to developer math models functions. Details about how the simulation executable calls these jobs (order, priority, frequency etc.) will be explained later.

The following diagram shows the relationship between the simulation definition and the math models.

Figure 9 Simulation Definition/Math Model Interface

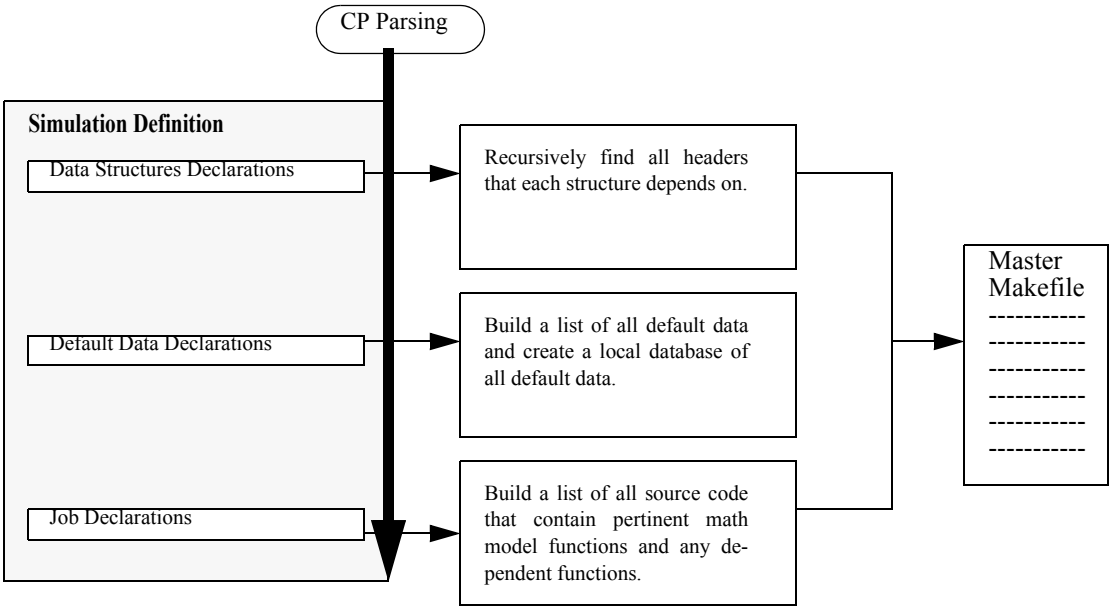


The simulation definition file points to a list of math model data structures, default data files and math model functions. With this information (and some special syntax in each header and source code file) CP is able build the simulation.

2.2.2 CP Processing

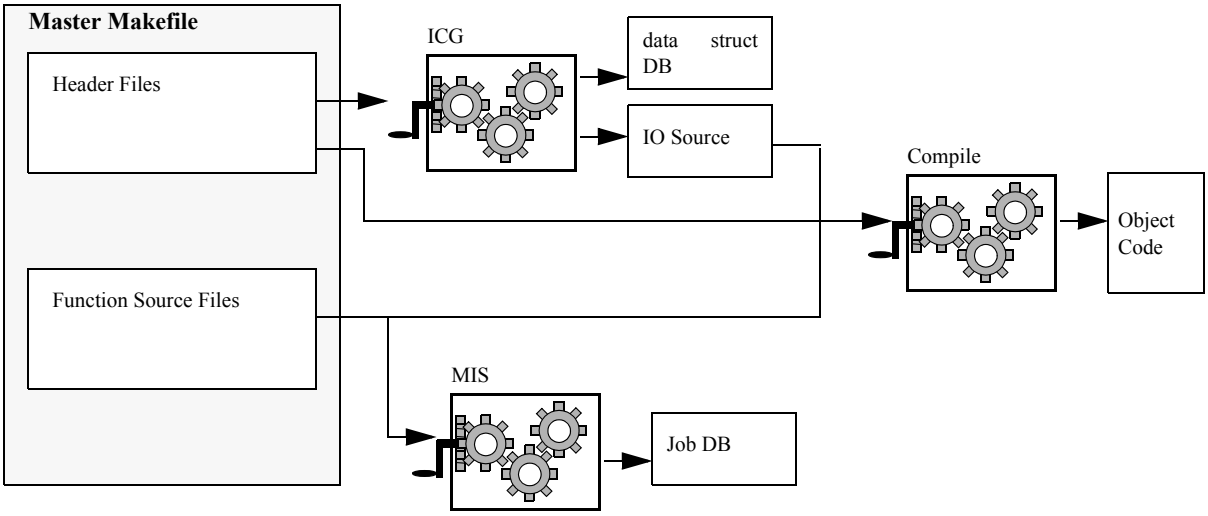
As CP parses the simulation definition, it builds source code, databases and a master UNIX makefile containing all simulation dependencies. The Master makefile contains all rules/dependencies for building the simulation.

Figure 10 CP's Processing Of Simulation Definition



Once the master makefile is generated, processing of all the code may begin. The master makefile makes calls to CC as well as the Trick processors, ICG and MIS, to process math model source code. ICG is responsible for parsing headers and producing a data structure database and IO code that will allow Trick to input and output to math model variables. MIS is responsible for processing "function" source code. It's primary job is to build a catalog database of math model jobs (or functions) with information about functions and their arguments. The figure below depicts how the master makefile, ICG and MIS work to produce object code.

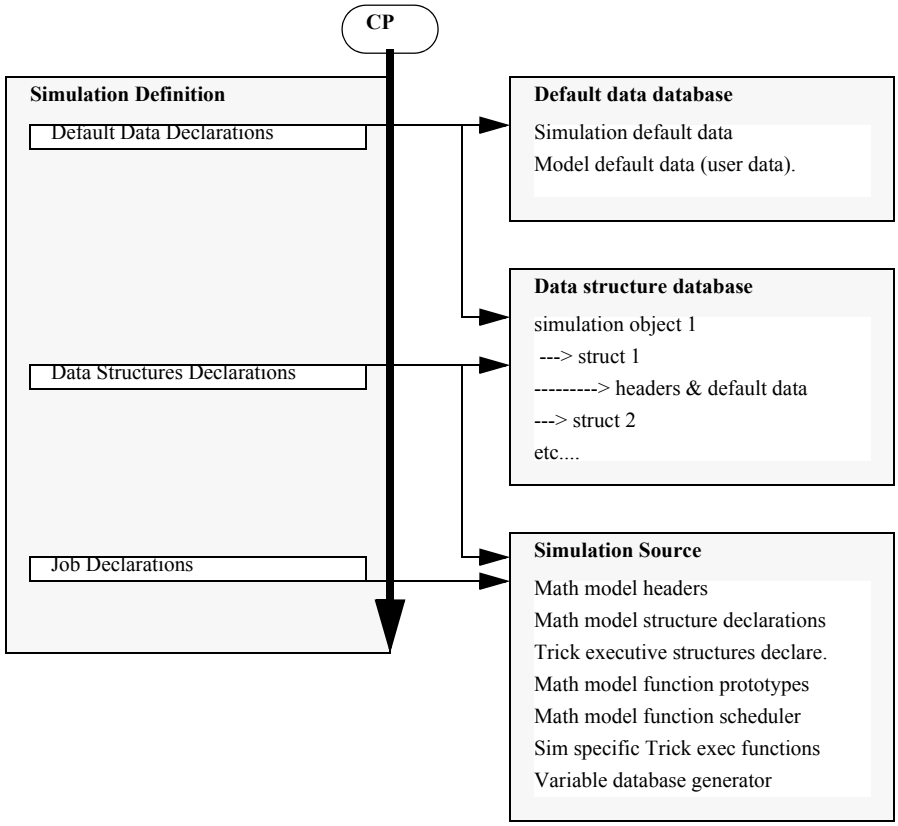
Figure 11 Master Makefile



2.2.3 Database And Code Generation

CP incorporates developers math models, data structures and default data to produce a simulation. CP also manages the dependency tree of simulation source code in a database (catalog) efficiency sake. It even produces a "documentation" HTML file for viewing the contents of a simulation. It is also responsible for creating a source file that contains logic for job order, job classification, job arguments, and run-time execution. This source file (S_source.c) is eventually compiled and linked into the simulation executable.

Figure 12 CP Database And Code Generation



2.2.4 Summary

CP is the interface between the developer and Trick. It is the processor that pulls together a simulation executable for the developer. The other two Trick processors (ICG & MIS) that Trick uses will be discussed in the following sections.

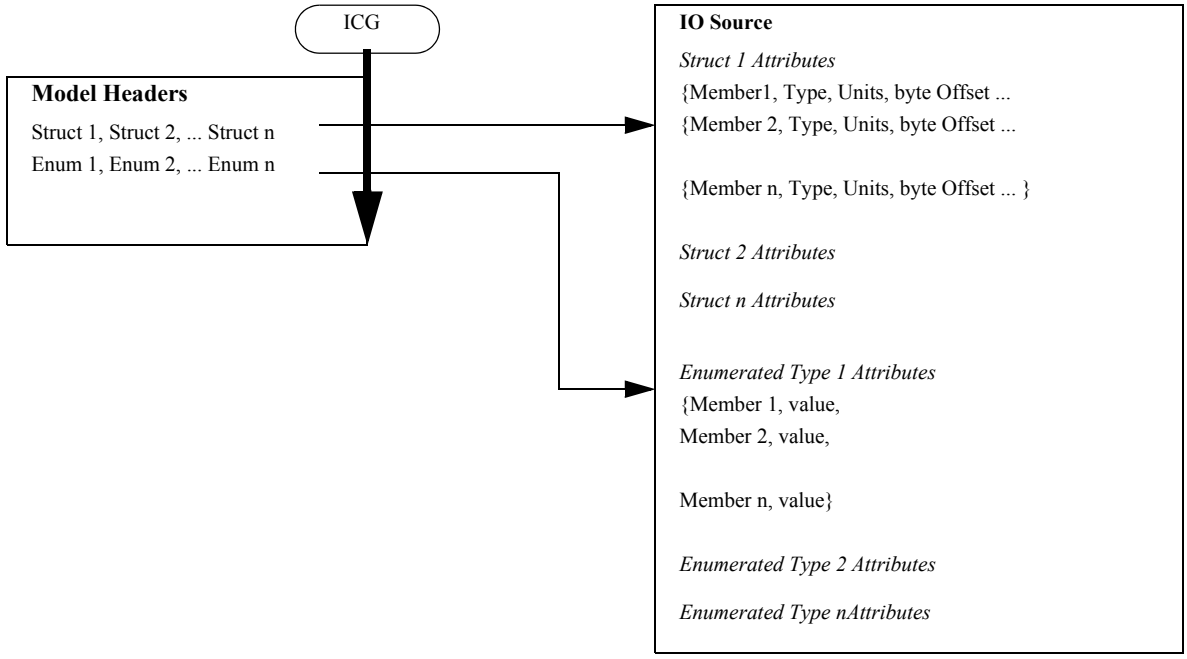
2.3 Interface Code Generator (ICG)

ICG parses developer created data structure header files and generates runtime executive Input/Output (IO) source code as well as database entries for all C struct, union, typedefs, enumerated types, and FORTRAN 90 types parsed (Note that F90 is only supported for SGI IRIX 6.5 and will be phased out). The source code generated is eventually compiled into a simulation which uses the types parsed. The type databases are used by ICG and by CP for data structure compatibility checks and for data structure instantiation in the simulation.

2.3.1 IO Source And Attributes

ICG is responsible for creating IO code from header files. IO code is the code that Trick uses to access math model variables in memory. Each structure and each enumerated type in math model header is processed by ICG resulting in what are called "Attribute" structures. Attribute structures contain all the information that Trick needs about a variable. Some of that information, like the Unit specification, is given by the developer in comments next to the variable in special Trick "comment" syntax. Other information, like each parameter's byte offset, is generated and used by Trick for accessing data.

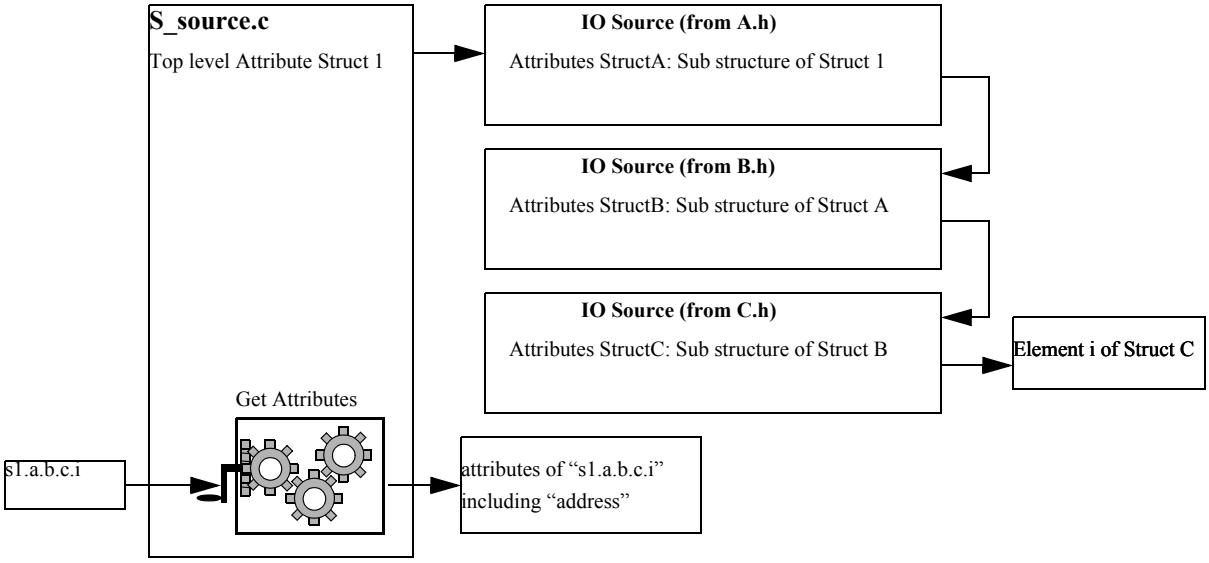
Figure 13 IO Source



The ICG generated IO source code and "attribute" data is eventually compiled and included into the simulation by the CP process. This IO source and attribute data allows the simulation "user" to input data through Trick's input processor with a "name = value" syntax. It also allows Trick's logging functions to output "user" specified variables through a data driven interface by specifying what variables are to be logged. This attribute data design also allows math model developers access to any ICG processed data structure variable by "name". Figure 14 below, shows the "attributes" concept for a data structure variable S1.a.b.c.i where "i" is the variable nested in structures S1, a, b, and c.

CP also uses ICG generated data to generate (in S_source.c) one "large" data structure that includes all math model and executive data. Math model function access to data structures are made through data structure pointers which are passed to functions that are specified in the simulation definition file and the generated simulation source.

Figure 14 Attributes Tree



2.3.2 ICG/HTML Auto Documentation

Another responsibility of ICG is creating documentation. ICG parses the comment sections of the variables and creates HTML pages that describe all processed data structures. In the end CP creates a master HTML page that links all documentation for the simulation together. Information about variable type, input/output specifications, unit specs, etc. are generated by ICG and displayed in the HTML documentation.

2.4 Module Interface Specification (MIS)

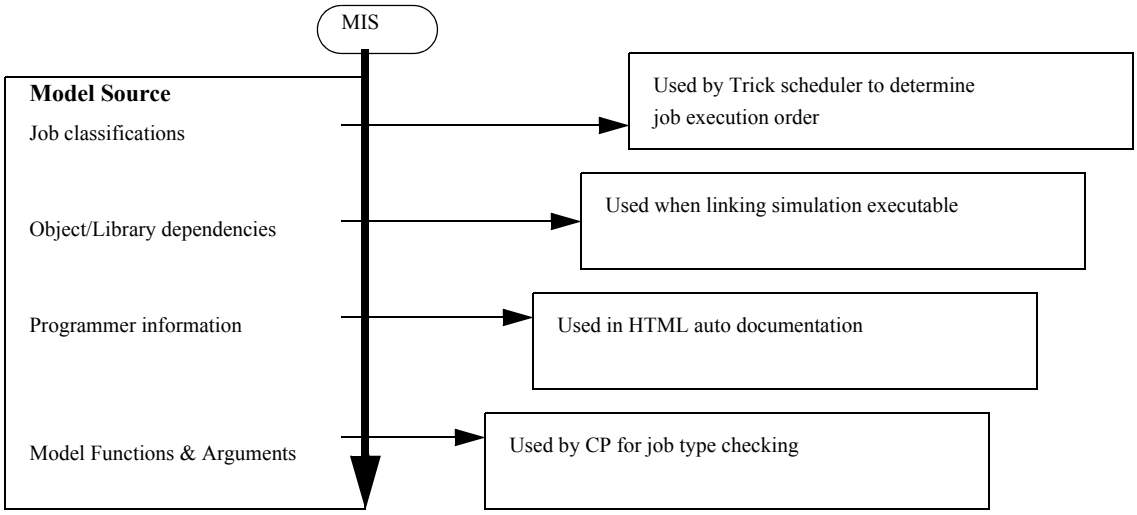
MIS parses developer created source code "function" files and generates a database entry for each module (model function) parsed. The module database is used by the CP for module classification and calling argument consistency checks as well as automatic code generation to hook the modules into the Trick executive.

MIS must process all source code module files before they can be integrated into the simulation by CP. This is primarily to ensure that the object code link list is constructed properly by the CP (object code dependencies are included with the module database information).

Special syntax is used in comment sections of the source code so Trick will know the jobs class, object dependencies, library dependencies among other things. Consult the User's Guide for details on syntax, available job classes etc.

Like ICG, MIS also creates HTML documentation that describes each module in the simulation.

Figure 15 MIS Functions



2.5 Make_build and Catalog

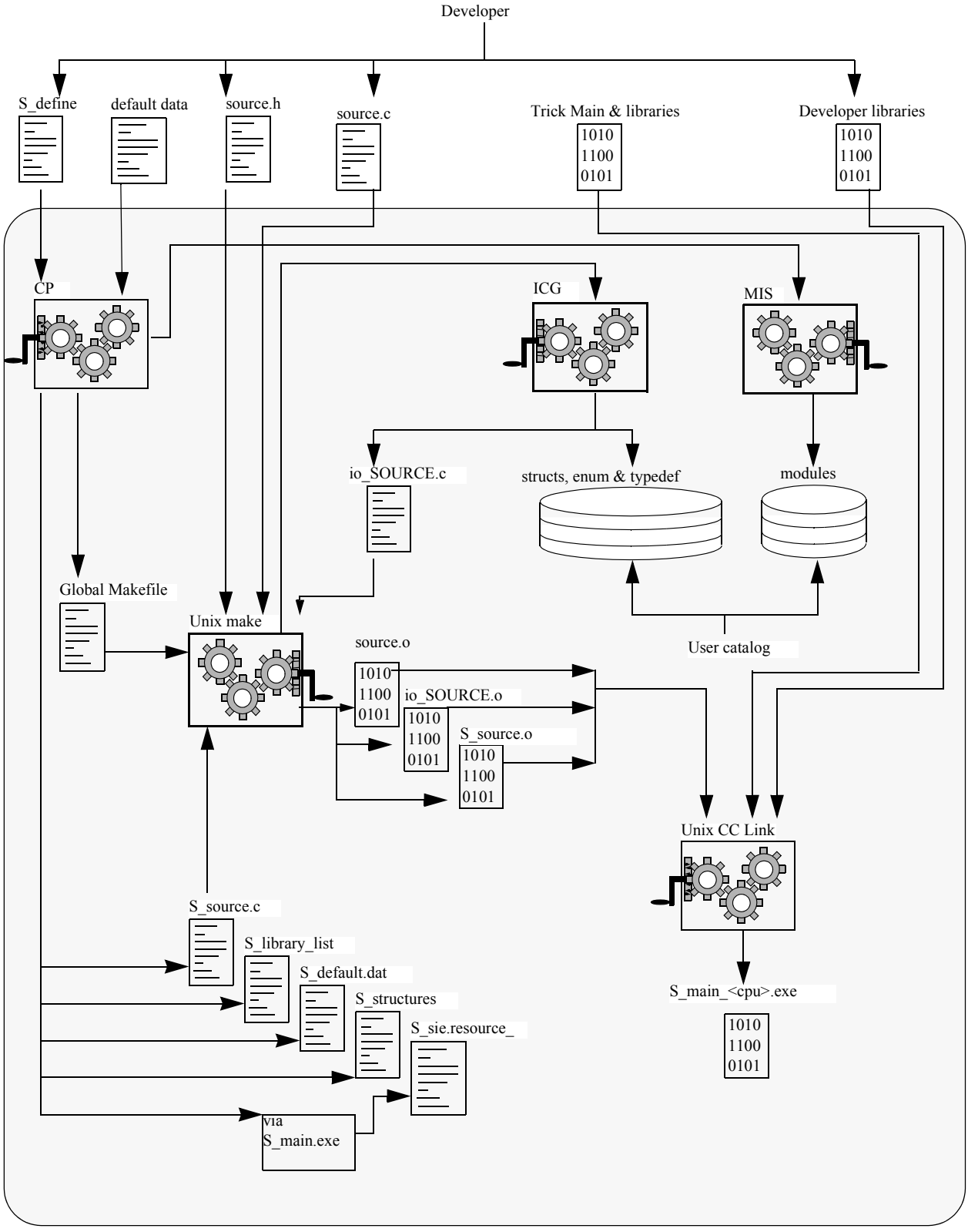
The Trick processors “make_build” and “catalog” have not been mentioned yet, but they are also part of the simulation building scheme that makes up trick. When executed from a math model directory, Make_build generates a UNIX Makefile for math model functions and header files. Make_build uses the X Windows utility “makedepend” to generate all make dependencies. Make_build is used by developers for “distributed” low level model development, but it is not required once CP is invoked and the simulation Makefile is generated.

The catalog utility servers two purposes for Trick developers. The first is when it is used as a reporting tool to list function and data structure database information for math models that have already been processed by MIS and ICG. The second purpose is to provide a catalog “initialization” and “building” capability to Trick developers. Each Trick developer maintains their own catalog directory environment which from time to time may require initialization and rebuilding. The catalog utility does this by scanning their environment setup and the math models that have been previously processed by MIS and ICG.

2.6 Putting it all together

Once all math model source code, with all the special Trick comments, is processed by CP, MIS, ICG and make, out comes a simulation executable. The following figure gives a grand picture for the interaction between all these pieces.

Figure 16 Trick Development Process Internals



3.0 The Simulation Executive

The Trick runtime executive is designed for both real-time and non-real-time applications which have time based (primary scheduling algorithm) and event based (secondary scheduling algorithm) scheduling requirements, including hardware-in-the-loop, and/or distributed (multi-processor and/or multi-box) applications.

The Trick executive is also designed to be data driven wherever possible. The runtime executive provides a complex array of user inputs which allow the user to configure the scheduling, real-time, multi-process, multi-box aspects of each application.

The Trick executive also has a built in capability for recording real-time performance data, and this capability is configurable through user input.

3.1 Real-Time Processing

Trick provides a real-time processing capability which is different from most real-time simulation capabilities. The Trick executive is a time based executive and can run in a non-real-time mode just as easily as a real-time mode. This is because Trick guarantees job execution orders and allows the developer to design guaranteed interfaces which are not effected by the execution time required for any one or more jobs. Frame based scheduling executives typically have problems handling real-time overruns because the frame pulse is the scheduling cue. The frame-pulse in Trick is a mechanism to monitor and maintain the real-time status of the simulation, NOT a scheduling mechanism.

An executive for a real-time simulation must guarantee that the simulated time matches the real-world time at specified intervals (real-time frame length). If the simulated time is greater than the real-world time, an overrun has occurred and must be dealt with. Trick's main or "parent" executive process does all job scheduling and real-time frame processing. Real-time frame processing is guaranteed by one of two methods; software time checks, or with operating system interval timer signals (itimers).

For both methods, the software time checks are performed at the end of each real-time frame by calling an operating system function to get the real-world time, and then comparing the real-world time to the simulated time. Although efficient, the software check alone has two major drawbacks for some real-time applications. First, it can not guarantee that an overrun will be detected when the real-time frame has elapsed; e.g. the software check will never stop an infinite loop. Second, since the software check burns the CPU during the underrun, it cannot will not go idle to let other processes have the CPU.

The software method can be combined with an itimer (SIGALARM) signal to guarantee overrun detection and handling and provide a process "wake up" or interrupt mechanism to facilitate CPU resource sharing. When itimers are used, the executive will pause (go to sleep) at the end of the real-time frame until the itimer signal handler receives the operating system SIGALARM interrupts at a specified interval. Once the signal is received, the handler performs checks on the execution status of the current real-time frame.

For both methods, if overruns occur which exceed the maximum overrun amount or the maximum number of overruns, the parent initiates a quick and graceful shutdown or it can go into freeze mode. Note that the executive does not know of the overrun condition until 1) it has finished its regularly scheduled real-time frame processing, or 2) the itimer signal is handled. This means that the overrun detection is not instantaneous, but dependent on synchronous itimer checks and indeterminate real-time frame completion.

When real-time frame under-runs occur, the executive can be configured (through input flags) to pause (uses no CPU resources) and wait for a SIGALARM interrupt, or to perform a spin loop (software only mode) continually checking the system clock against the simulated time until the "real world" system clock "catches up".

Through the input file, the user can set over-run limits, use software time checks and itimers, and use real-time process control features (locking the process in memory, assigning and locking a process to a processor, setting the process priority, running a simulation time to real-time ratio other than 1 to 1, etc.).

3.2 Source Code Architecture

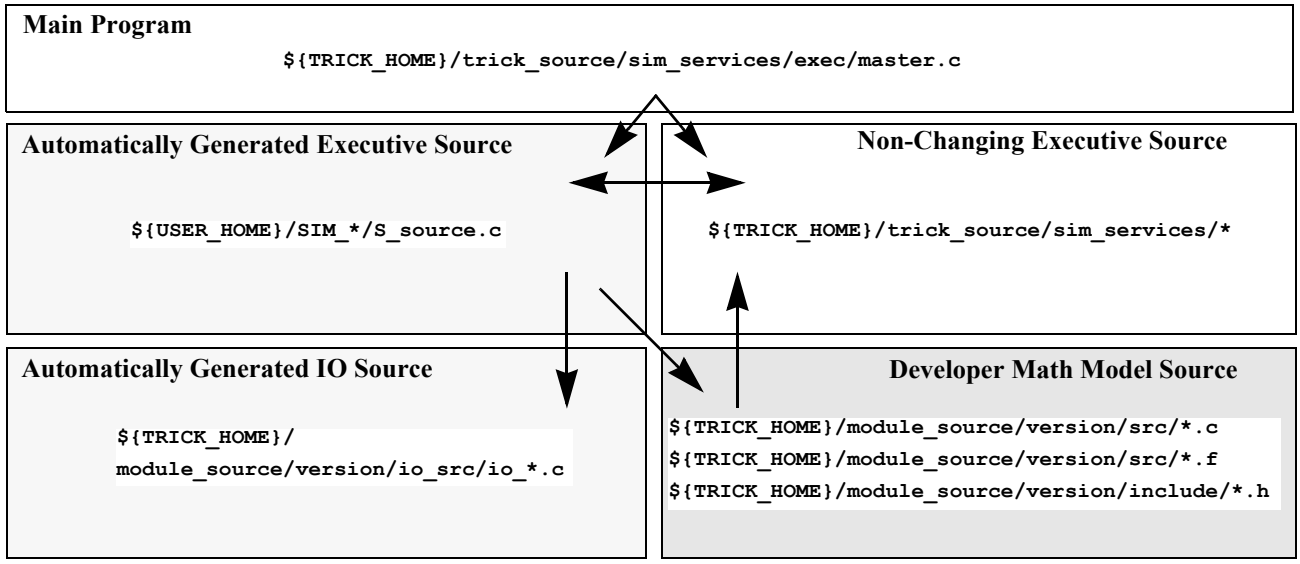
What source code comprises the simulation executive? What Figure 16 does not entail, this section covers.

A complete simulation source code body is comprised of the following components:

1. Trick runtime executive `main()` program (`master.c`),
2. Trick runtime executive source code which remains unchanged regardless of the simulation application (`${TRICK_HOME}/trick_source/sim_services/*`) (Note that the main() program also remains unchanged),
3. automatically generated Trick executive source code which is specific to each simulation application (`${USER_HOME}/SIM_*/S_source.c`),
4. developer generated math model source code (`${TRICK_HOME}/module_source/version/src/source.c`, `${TRICK_HOME}/module_source/version/include/source.h` and `${TRICK_HOME}/module_source/version/include/source.d`), and
5. automatically generated source code for math model runtime IO (`${TRICK_HOME}/module_source/version/io_src/io_*.c`).

The relationship between these components is shown in Figure 17 which depicts a typical simulation source code architecture. The arrows in this figure indicate function calls; e.g. the main programs make function calls to the auto generated executive source code and the non-changing executive source code. The three levels depicted in the figure represents high, intermediate, and low level function calls. The source code blocks are shaded to separate the non-changing executive code (clear), from the automatically generated code (lighter shade), from the developer generated math model code (darker shade).

Figure 17 Runtime Source Code Architecture



The Trick main programs are contained in the `${TRICK_HOME}/trick_source/sim_services/mains` directory. The main programs are in their own source code directory because all the other executive source is packaged in archive libraries and the main program objects cannot be in the libraries if the simulation compile is to link properly. `master.c` is the parent runtime executive which handles initialization, job scheduling and real-time processing control.

`master.c` calls functions from both the auto generated and non-changing executive source code. `S_source.c` is generated by CP for each simulation. `S_source.c` contains the high level runtime IO functions which call the lower

level IO functions (`io_*.c`) and the developer supplied math model jobs. The non-changing executive source code also provides some useful low level services for developing math models.

3.3 Memory Architecture

The Trick executive uses good old malloc-ed memory for 100% of its global data (ALL data structures) requirements. This design is possible since Trick uses threads rather than forking off children. This design has the benefit of being able to be "purified" by the Rational Purify tool.

In a multiprocess group (Master/slave) set up, data is shared by importing and exporting data over sockets. There is no "shared" arena.

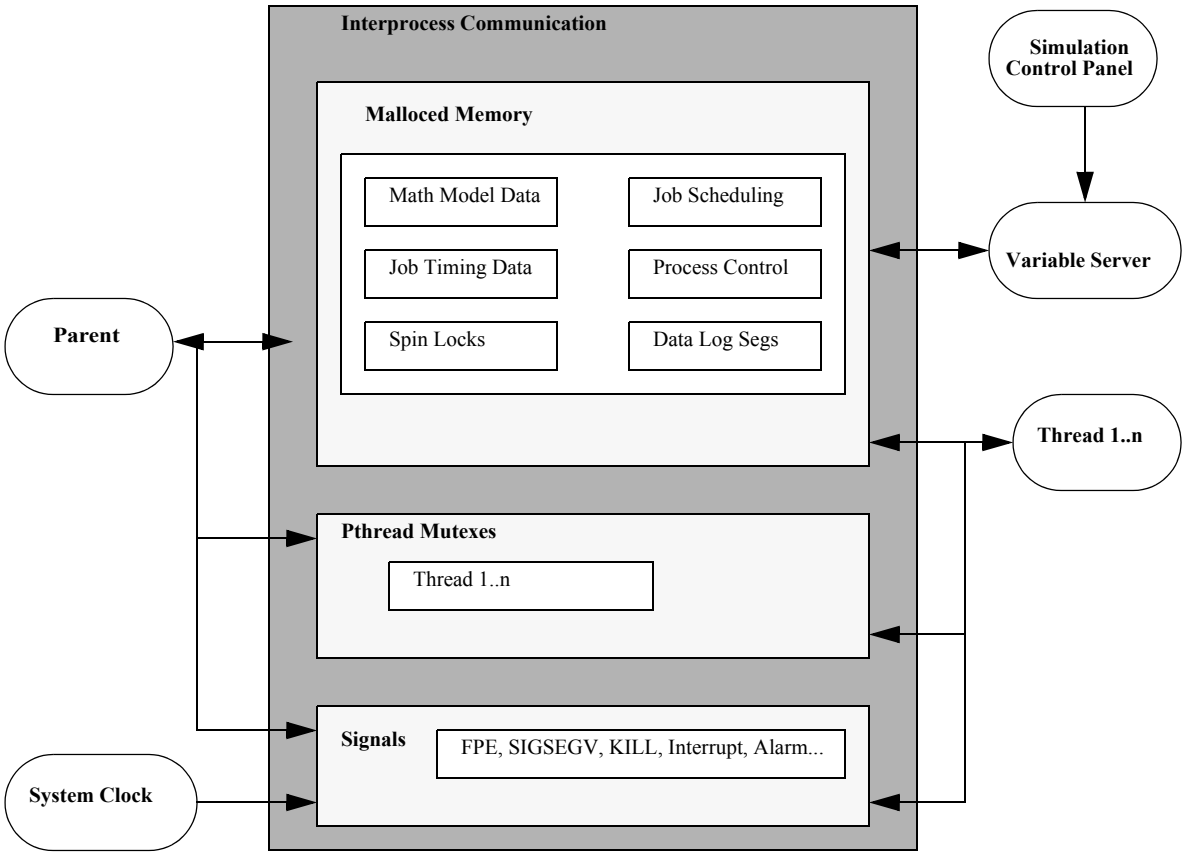
3.4 Variable Server

Trick offers a way for an external process to get/set math model variables in memory. There is a thread called "variable server" that has access to memory. The variable server accepts clients and gives them access to simulation data and allows clients to set data to specified values as well. The simulation control panel is an example of such a client.

3.5 Process Architecture

Trick simulations can be configured to be single process or multi-process, including multi-computer. The S_define file syntax and data driven input parameters control this configuration. A simulation unit defined with a single S_define file is referred to as a "Process Group" or PG. This PG can be single process or multi-process (multi-process within a PG refers to UNIX threaded processes that run in parallel). Figure 18 shows the Trick executive interprocess communication (IPC) architecture for a single PG multi-process simulation running on a single, possibly multi-cpu, workstation.

Figure 18 Single Process Group Architecture



Processes are represented by ovals and the shaded box represents the ipc mechanisms. In this figure, the executive is represented by a parent executive process, a simulation control (Tcl/Tk) process, and one or more threads that are spawned by the parent executive (according to specifications in `S_define`). In general, a multi-process simulation should only be executed on a multi-processor machine. If a multi-process simulation is executed on a single CPU machine, the simulation will run slower than the same simulation configured for a single process. This is due to the context switching that would be necessary on the single processor machine.

Malloced memory is used for all data structure data. Memory segments can be accessed by the parent executive *and* all threads. The parent executive controls all job timing, scheduling, and process control data; the child threads merely access the data. Math model data can be accessed or modified by either the parent or the child.

Pthread mutexes (or Spinlocks) are used to tell the child threads to start processing their respective job queues for each scheduling frame. The parent constructs all the child job queues and then sets a mutex for each child with jobs on its queue. Before the children receive their job queue start mutex, they are idle (using no CPU resources unless Spinlocks have been selected). The children receive and reset their respective mutex and start processing their job queue. The parent executive knows when the children have finished via job completion flags.

Sockets are used to managed health and status prints (debug, status, error message, etc.) from the parent executive and child processes to the Simulation Control process.

Signals are used to manage process termination as well as real-time synchronization. The typical termination signals INTERRUPT, KILL, FPE (floating point exception), SEGV (segmentation violation), and BUSERR (bus error), are all trapped so that the simulation can start a graceful shutdown. The ALARM signal is used in the real-time mode to synchronize the simulation clock with the real-world clock (the computer system clock).

3.6 Executive Loop

The heart and soul of the simulation executive is shown in Figure 19. The two figures (Figure 20, Figure 21) that follow cover lost detail from Figure 19. Whether the simulation is setup as a real-time or non-real-time simulation, the control flow for job processing shown is the same, and controlled by the parent executive process. However, processing for single and multi-process simulations could be slightly different because of the parallel processing and could effect simulation output. Job dependency mechanisms are available to ensure that a multi-process simulation will produce identical output to an associated single process simulation.

Figure 19 Executive Loop

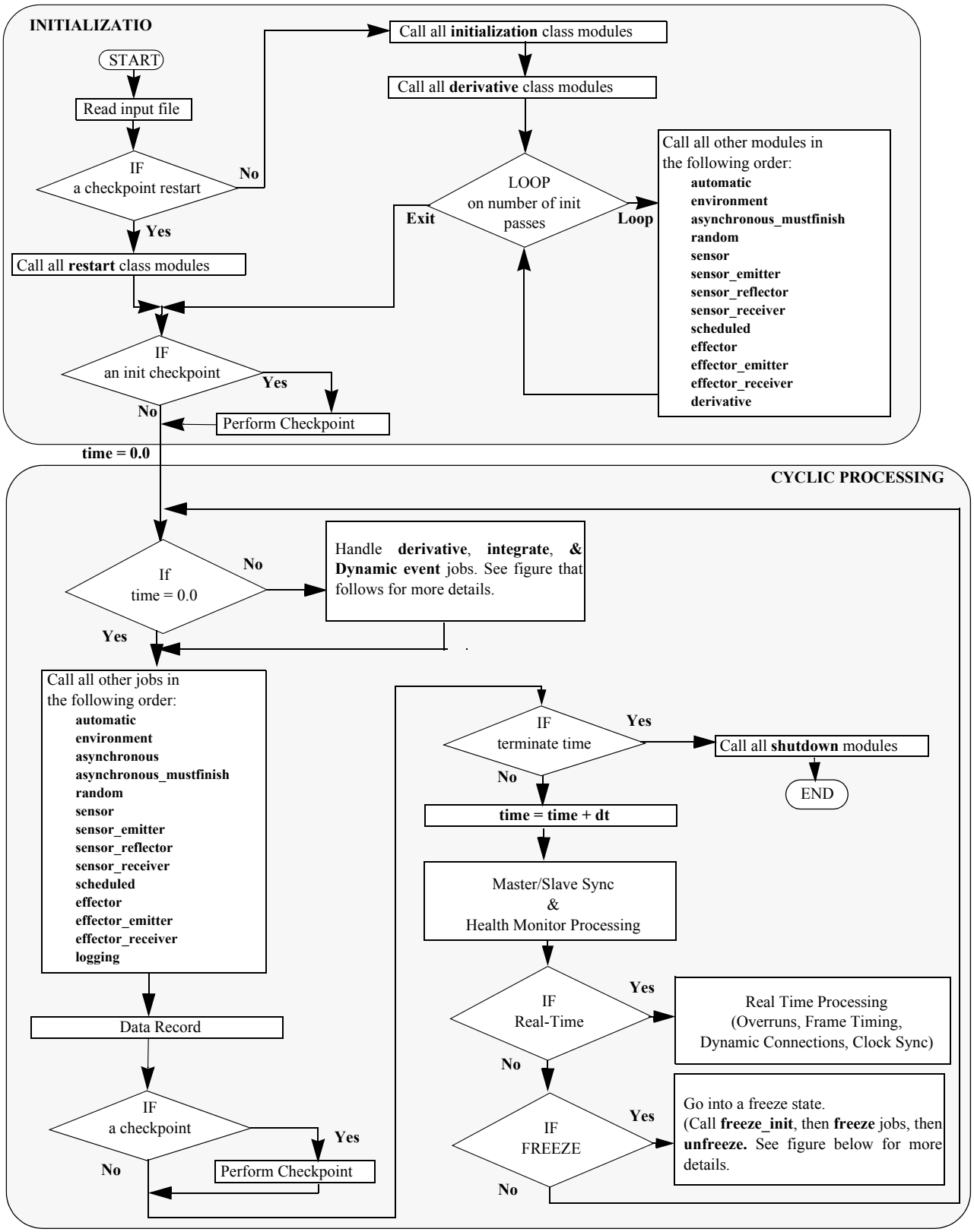


Figure 20 Derivative/Integration/Dynamic Event Control

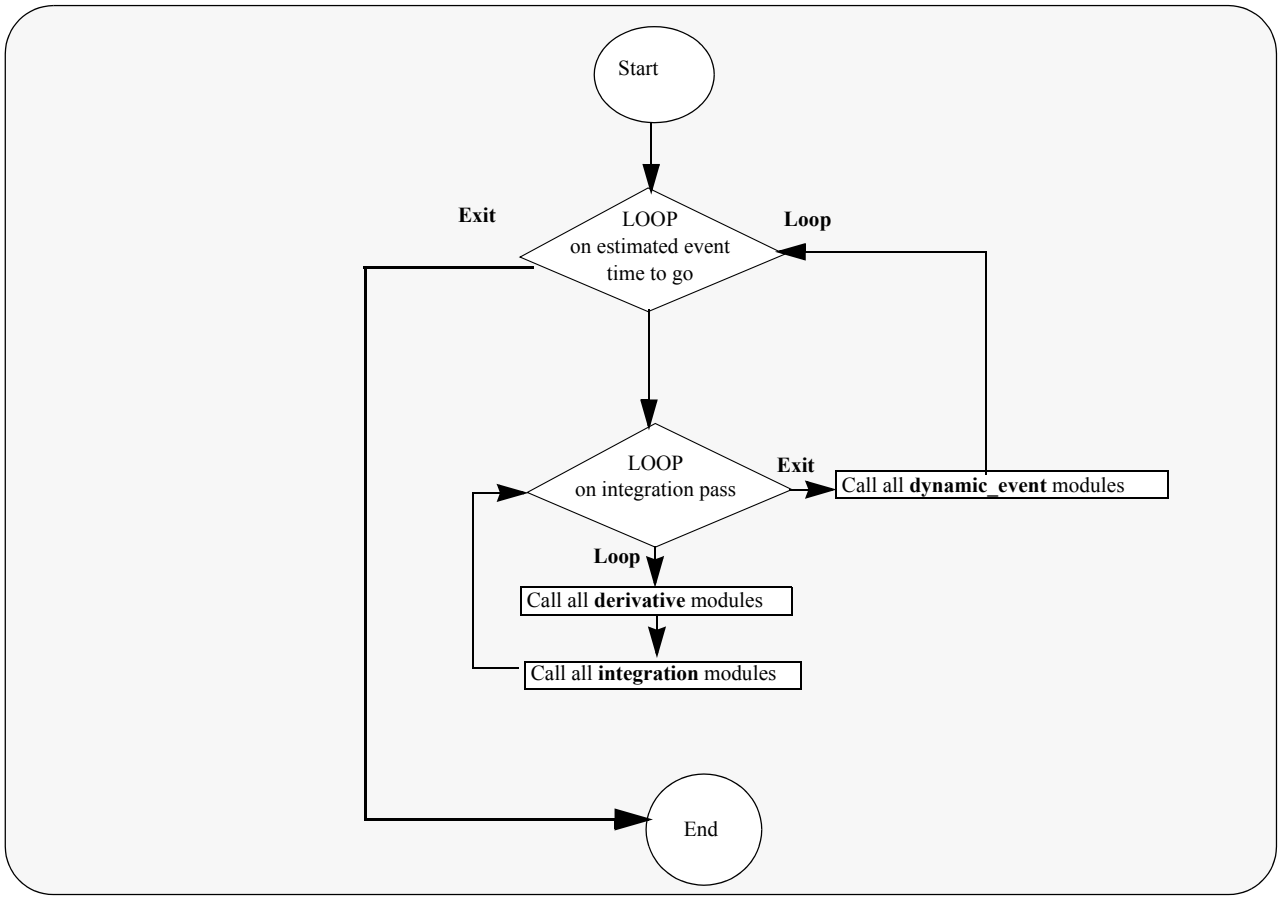
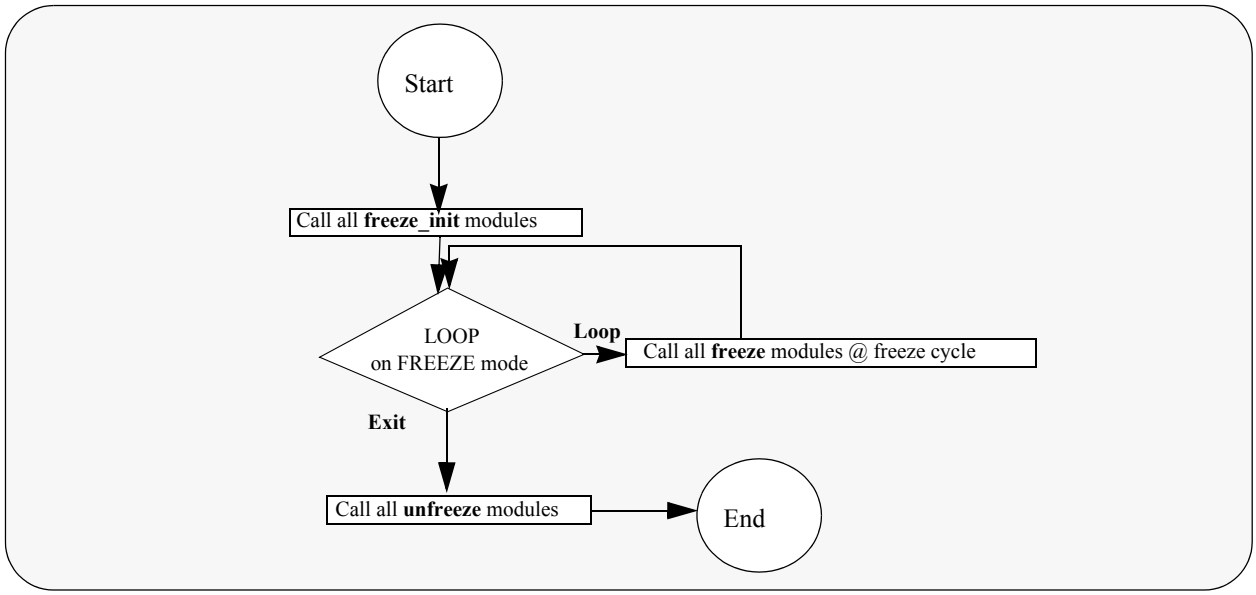


Figure 21 Freeze Control



3.7 Parent/Child Thread Details

A multi-process simulation requires more attention from a developer because data flow is not guaranteed unless `depends on` statements are added to the `S_define` file to force a critical data path. The main executable in a multi-process simulation is called the parent process, and all processes spawned by the parent are called child threads. All `initialization` jobs specified for a child thread will execute in that child thread (as opposed to executing in the parent process), but are guaranteed to execute in the order specified in the `S_define` file, in other words, there is no parallel processing during initialization. Jobs of other classifications will run in the process specified within the `S_define` file.

The executive will build the main job queue, which should be identical to the queue for the single process simulation, but the executive will also build separate jobs queues for each process. If no `depends on` statements are specified in the `S_define` file, then the executive will immediately start executing the first jobs on all process queues in parallel. A `depends on` statement will set a dependency between jobs which will force the executive to hold off the execution of one job (on any thread) until the jobs it depends on have completed. Even if all jobs in the parent process queue have been completed, the executive will not continue to the next job scheduling cycle until all jobs scheduled for the current scheduling cycle have finished. The exceptions to this rule are the `asynchronous` and `asynchronous_mustfinish` jobs. `Asynchronous` jobs are scheduled by the executive, but the executive does not wait on these jobs to finish before continuing with normal job scheduling. Instead, the executive will check to see if it has finished and then start it again on its next cycle. `Asynchronous_mustfinish` jobs are similar to `asynchronous` jobs, except that the executive will "wait" for the asynchronous_mustfinish job to finish when it is time to restart it again.

Processes can be configured to block on job queue starts and job dependencies using either spin locks or pthread mutexes. Spin locks should not be used if the multi-process simulation is running more processes than available processors. Spin locks are implemented using a busy loop checking a flag in shared memory - this equates to 100% CPU utilization for the active process. If the simulation has fewer processes than processors, spin locks provide a more responsive (smallest latency) method for blocking the various simulation processes. Pthread mutexes, are good when there are fewer processors than processes. Mutexes allow the blocking thread to release the processor so that other threads can use the processor. Although mutex overhead is higher than spin locks, mutexes provide the best performance when the number of processes is greater than the number of processors.

3.8 Executive Timeline Example

Let's step through a real-time simulation multi-process job scheduling scenario to help visualize job scheduling. (Figure 22) If an S_define had **only** the following scheduled job entries and the simulation was running with a 0.5 second real-time frame:

```
sim_object {

        (0.1) test/v1: test_funct();
    C1  (0.2) test/v1: test_funct1();
    C2  (0.4) test/v1: test_funct2();
    C3  (1.0) test/v1: test_functasync();
    C4  (1.0) test/v1: test_functasyncmf();

} test;
```

the parent executive and child threads1 an 2 would wait for all three jobs to complete from the time 0.0 job frame (all scheduled jobs that have a start time of 0.0 will be called at time 0.0) before test_funct() would be called at the 0.1 job frame. Note that child threads 3 and 4 have asynchronous type job classes on them, so they will not be synchronized with the other job frames. The next job frame would be 0.2 in which test_funct() and test_funct1() would be called in parallel. Then the parent executive would wait until test_funct() and test_funct1() from the 0.2 job frame have completed before test_funct() would be called at the 0.3 job frame. The parent executive and child thread would then wait for test_funct() to complete and then call test_funct(), test_funct1(), and test_funct2() in parallel for job frame 0.4. Then the parent executive would wait until test_funct(), test_funct1(), and test_funct2() from the 0.4 job frame completed before waiting again for real-time to catch up to the simulation time. This final waiting time is labeled as the real-time frame "underrun" time. (see Section 3.1).

Finally test_funct() would be called at the 0.5 job frame when the real-time clock reached 0.5 seconds. This job frame and real-time synchronization policy would repeat through out the simulation.

The second mechanism that the executive uses for the mutual exclusion of shared data of jobs running in parallel is the S_define "depends on" syntax mentioned in the previous section. This mechanism prevents jobs executing in parallel on a given job frame from clobbering each other's data in the reading and writing of common data structures. For example if an S_define file had the following depends on entry to go with the above jobs:

test:test_funct() depends on test:test_funct1() The executive would force Parent process to wait for the completion of the test_funct1() job scheduled on C1 before calling test_funct() in any common job frame (0.0, 0.2, 0.4, etc.). If we apply this dependency to the timeline example that we have been working with, it will cause real-time overruns (Figure 23).

Figure 22 Executive Job Frame Scheduling Timeline Example

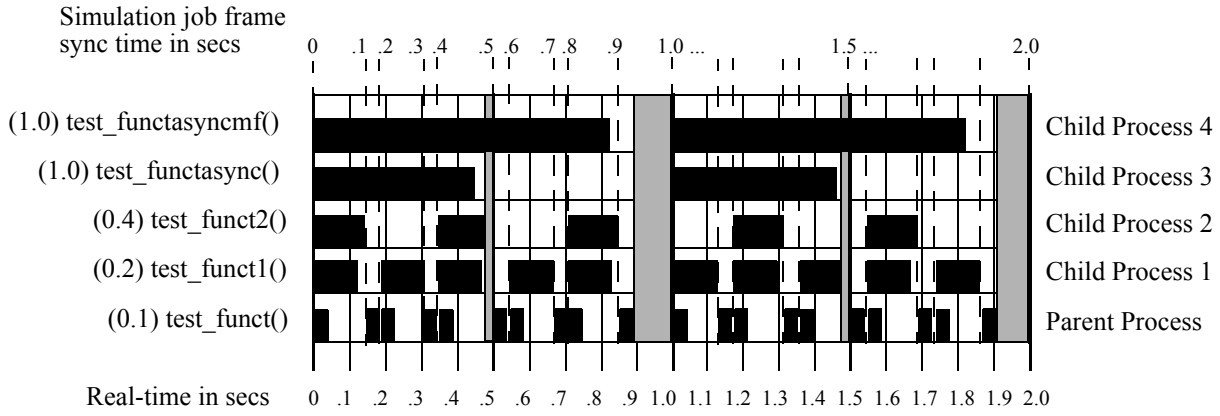
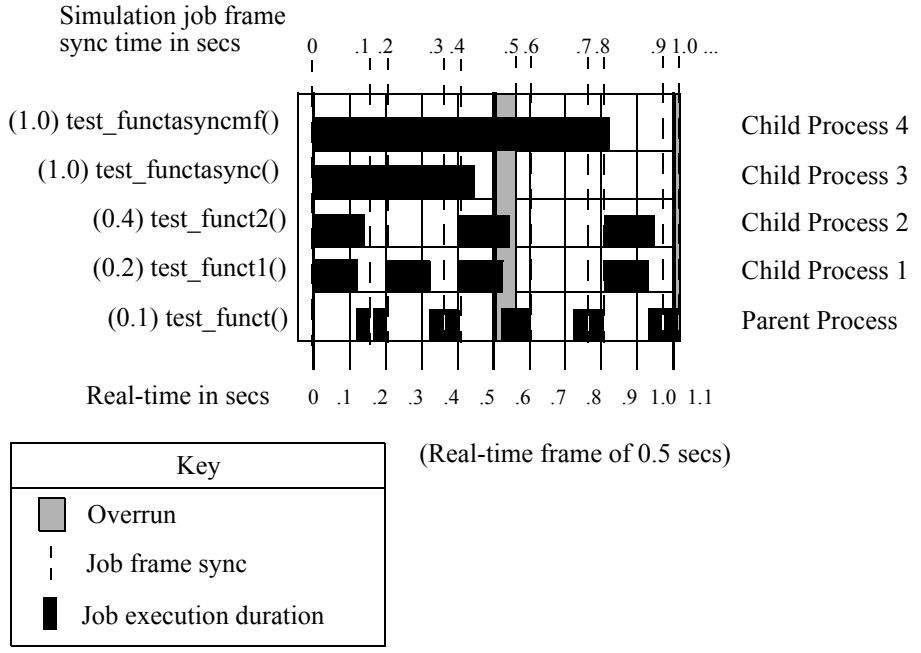


Figure 23 Executive "depends on" Job Frame Scheduling Timeline Example



3.9 Multiple Process Groups (Master/Slave)

The Trick executive has the capability to synchronize the execution frames (real-time or non-real-time) of simulation Process Groups (PG) running on the same or different processors/computers The multi-PG synchronization scheme can be visualized as a software implementation of an external interrupt generator, but with more capabilities. Since the synchronization is performed via software, frame overruns experienced by any of the simulation executables can be handled gracefully by all the executables without loosing real-time (without skipping a real-time frame) AND with guaranteed data path integrity; characteristics which a system driven by a hardware external interrupt cannot achieve easily.

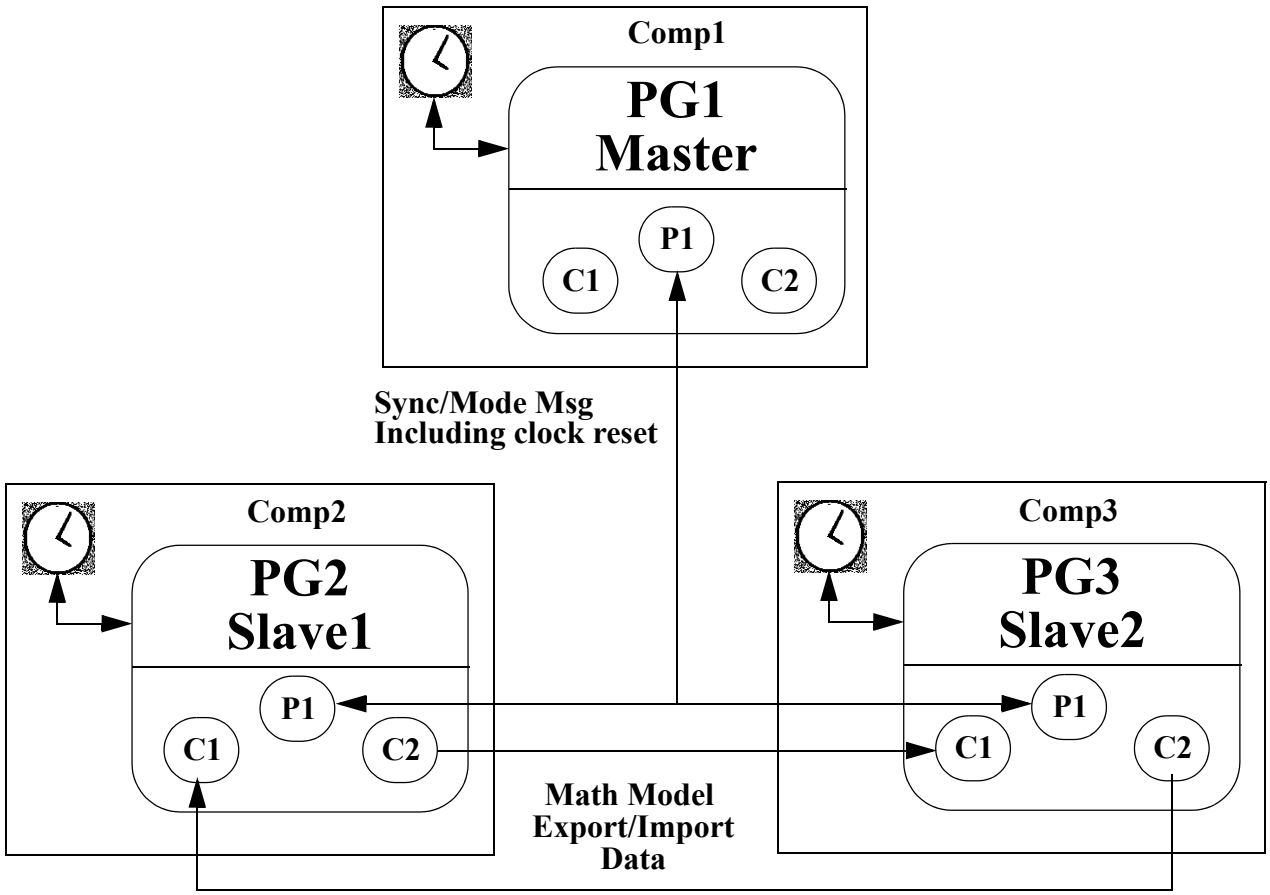
In the multi-PG scenario, one PG is considered the master PG and all other PGs in the simulation are slaved to the control of the master PG. The master is responsible for generating synchronization queues to the slaves, collecting frame execution status from the slaves, controlling the simulation mode (initialization, run, freeze, shutdown), and synchronizing the slave clocks with the master's clock. Synchronization communication between the simulation executables is performed via sockets (TCP/IP). The identification of the Master PG is done through the input file, and since the Master starts the Slave PGs, slave PGs do not have to be identified through input. This data driven capability provides a flexible configuration scheme for Master/Slave designation.

Simulation data packets can be passed between Trick PGs with Trick's Export/Import capability. With this capability, simulation designers can specify data structure packets to be exported and imported into the PG via the S_define file. CP will parse this syntax and then generate the communication code to export or import the specified data at the specified frequency of the designated job the export/import syntax is attached to. The identification of the host machines and PGs is done through the input runstream which again makes the distribution of PGs highly configurable. The export/import capability can also be used between parent and child processes within the same PG, but is not usually used because process group parent and children already share the same memory.

Figure 24 shows a Process Group master/slave communication scenario. The diagram depicts a multi-process master PG synchronized with two multi-process slave PGs. The diagram shows the Parent processes from each PG communicating

synchronization, mode and clock resets. The diagram also shows child process #2 in slave #1 exporting data to child process #1 in slave #2, and child process #2 in slave #2 exporting data to child process #1 in slave #1.

Figure 24 Synchronization and data communication between Process Groups

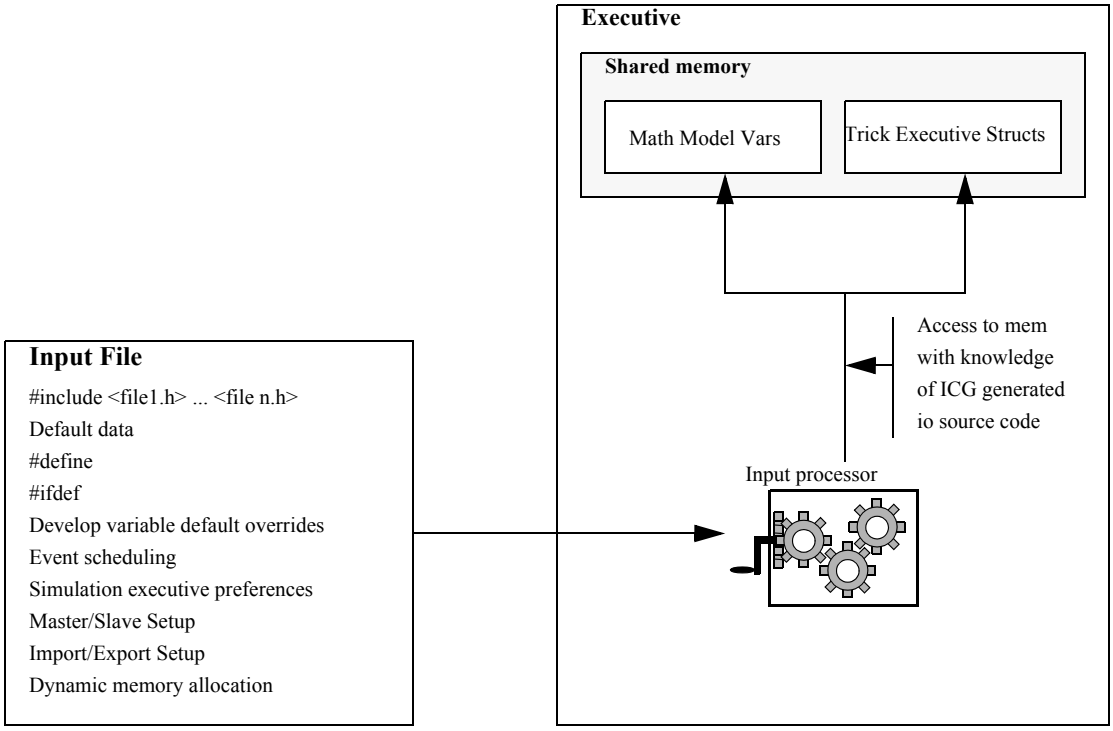


3.10 Input Processor

The simulation executive is designed to be data driven. The interface between user and simulation executive is the input processor. Users/developers create input files in an ASCII C-like syntax. Before a simulation hits its first initialization job the input processor parses user created input files. As it parses, it sets variables, loads default data etc. The input processor handles #defines, #ifdef, #includes etc. Very complex input scenarios are built using these constructs. Dynamic simulation time based events are also specified in the input file. Much of the simulation executive is customized through the input file. Simulation stop times, data recording parameters, control panel preferences, Master/Slave setups, interprocess group import/export specifications and a host of other items are setup through the input processor.

In order to access data in shared memory, the input processor must have knowledge of the IO source code (attributes) generated by ICG.

Figure 25 Input Processor



3.11 Data Recording

Data recording is built into the Trick executive. Data recording can be completely configured via the input file.

Data recording initialization, execution, and shutdown is automatically performed by the executive. Data recording is not a scheduled job like the input processor.

Users need only to set up data in the input file to use data recording. Some of the more important capabilities involve the output format for recording, the output device for recording, and the recording frequency.

Data recording parameters (variables logged) are organized into groups. Each group has its own data format, data frequency, destination and frequency.

3.11.1 Formats

There are three data recording output formats: binary, ascii, and fixed ascii. The binary format is a continuous stream of bytes which represent the data records, one after the other, without any special separators. The binary format is meant for post processing programs. The ascii formats are suitable for printing or viewing via a text editor.

3.11.2 Devices

Each format can be directed to a disk file, computer memory, or a printer. For all devices, the data ultimately ends up in a disk file. Recording directly to memory is much faster than recording to disk, but the memory resource is typically much scarcer than hard disk space. Once a simulation has completed, any data recorded to memory is transferred to a disk file. The printer option is only operational for initialization and termination records for ascii and fixed ascii output formats. For the printer option, the records are first written to a disk file and then spooled to the printer.

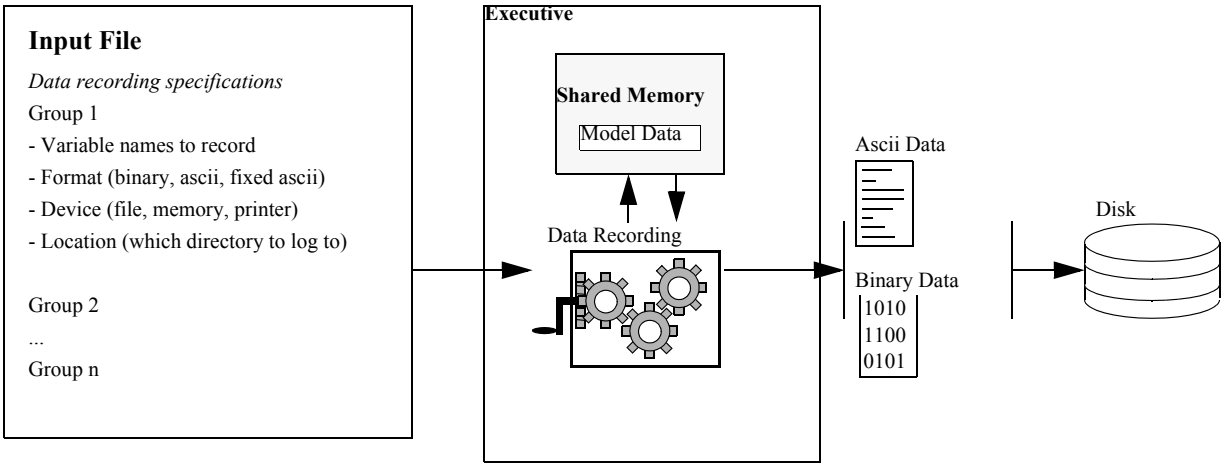
3.11.3 Output Destination

The user can specify an output directory to hold all data recording files generated by the simulation. If the user does not specify an output directory the `RUN_*` input directory is assumed. The data recording files include a binary data header file, and ASCII data header file, and files for every data recording group. The user can specify the name for the data recording manager for the simulation. This name is used to give the data recording header files (files which contain a list of all data recorded for the sim) a unique name. For users using a single simulation this name is unimportant, but for users using a multiple simulation application where all the recording data resides in the same directory, this name is important.

3.11.4 Frequency

Each data recording group can be recorded at different frequencies from the other recording groups. In addition, special recording frequency options are available. For all the frequency options, the processing for the option occurs at the cycle time specified for the recording group. Data can be recorded at every cycle, or data can be recorded only when the data has changed from the previous cycle, or, if data has changed, the data records on both sides on the change can be recorded (this creates an output for plots which looks identical to recorded every data point), or data can be recorded only for the initialization record, or only for the termination record, or only for both the initial and terminal records.

Figure 26 Data Recording

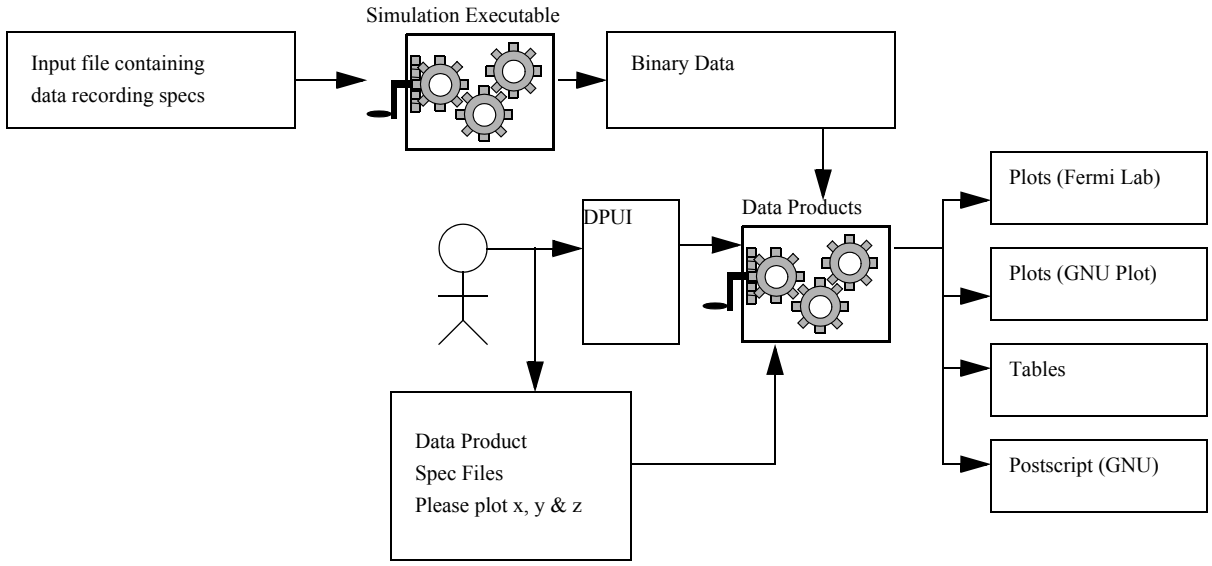


4.0 Data Products

The new Data Products was designed from scratch based on requirements collected from several engineering and operations customers. The requirements which drove the resulting design were 1) real-time updates from a running simulation, and 2) access to recording data across any number of recording groups.

Data products is a command line program. It is responsible for gathering data from binary files and feeding data into Fermi Plot widgets or GNU plot. There are GUIs that drive the data products for ease of use. The figure below gives the overall picture of how data products fits into the Trick picture.

Figure 27 Data Products Overview



4.1 Data

Data products works on data recording files already written to disk. It is a post processor. Data is written by a Trick simulation to disk in “data logs” by it’s data recording mechanism. Each data log contains: 1. Binary data 2. Header info for variable name and byte size of each variable logged. The header info is in text files. A collection of data logs is called a “Log Group”. Log groups may have many binary files and many headers associated with it, but will reside in the same directory. There is a well defined data products class that handles log groups called “LogGroup”.

There are many supported data formats. See the User’s Guide for details. To see the Trick binary format refer to the User’s Guide as well.

4.2 DP Specification Files

These are the files created by users so that data products will know what data and how to display the data once gathered. The syntax of the DP files may be found in the User’s Guide. The class responsible for handling the parsing and collection of knowledge of the DP spec file is called “DataProduct”.

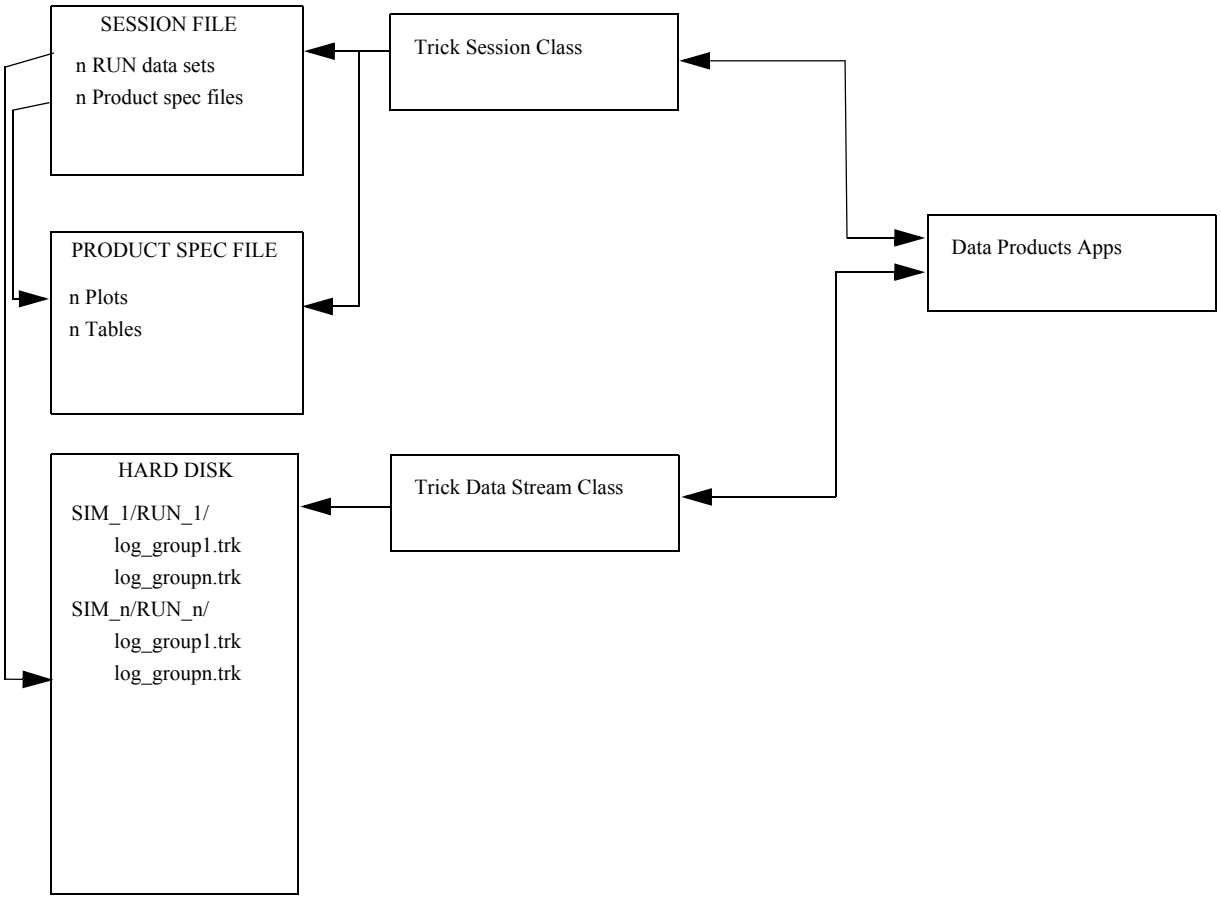
4.3 Session File

The Session file is the file that the plotting and table applications accept as an argument for creating the plots and/or tables. The Session file is basically a pointer to DP specifications files and Trick data. There is a class called “Session” that handles Session files.

4.4 Overall Architecture

By and large the Trick applications (fplot and table) use the session class, their own internal classes and the log class for accessing data. The diagram below shows this relationship.

Figure 28 Data Products Architecture



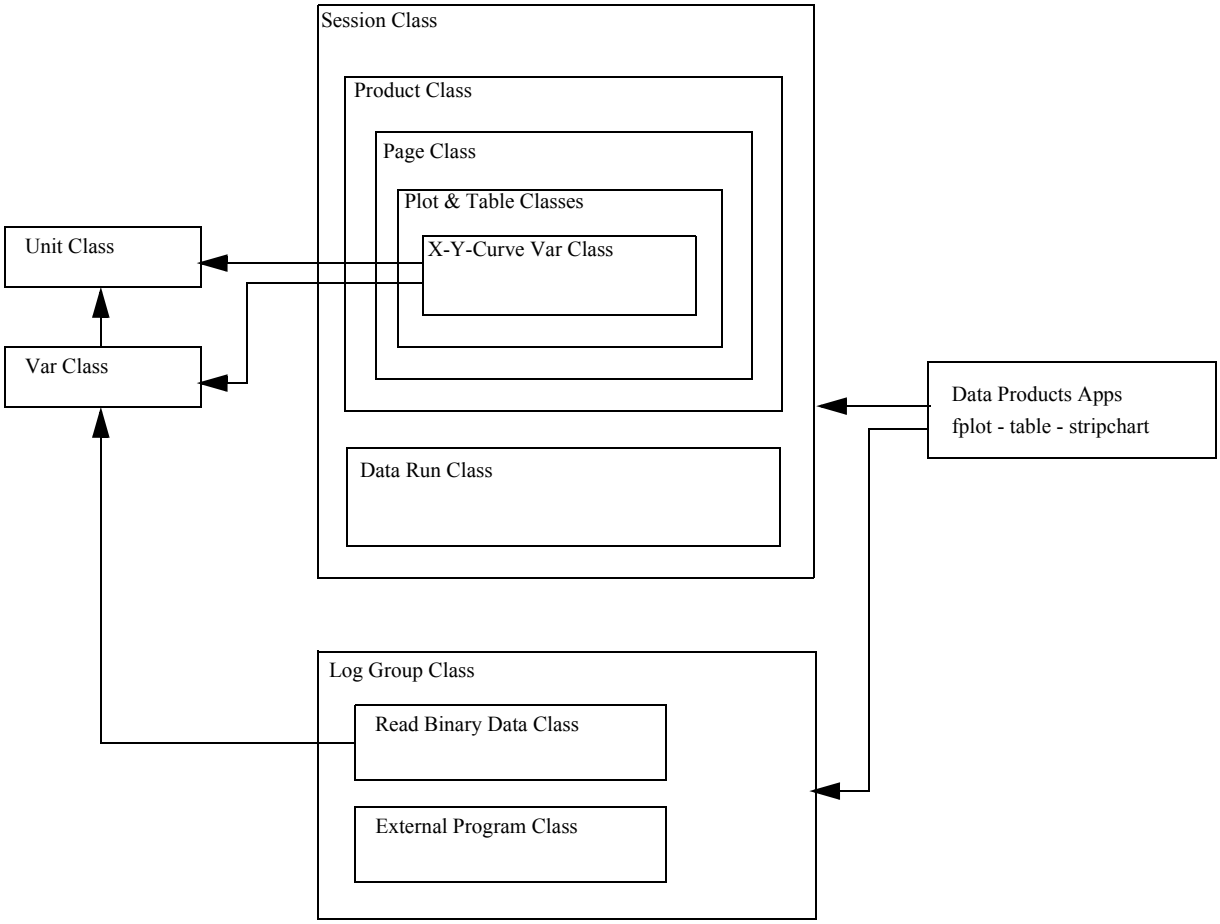
4.5 Class Architecture

Trick data products can be broken into five different areas.

- Applications
- DP Session class
- Log class
- Variable class
- Unit class

The following diagram shows how these classes are organized and utilized.

Figure 29 Class Architecture



5.0 Trick Environment

5.1 Developer Environment

Before a developer can begin development he/she must obtain a Trick UNIX environment. A script is used to “install” the developer into the Trick UNIX environment. Environment variables such as TRICK_CFLAGS must be set for the Trick utilities to be able to build a simulation. These mandatory environment variables are introduced into the developer’s environment via a resource file that he/she sources within his .cshrc or .profile file.

5.2 Run Time Environment

It is possible, but not necessary, to run a simulation without any UNIX environment variables. Trick manages this by keeping a list of defaults for all environment variables that it may need during the course of a simulation run. Variables are resolved by first checking the UNIX environment. If the variable is not found there, it falls back to an internal default. So, overriding a variable’s internal definition is as easy as setting the variable in the UNIX environment.

6.0 Monte Carlo And Optimization

Monte carlo is the process of iteratively calling a simulation over a set of predetermined or auto-generated inputs. Trick has designed its monte carlo capability to run distributed.

6.1 Master/Slave Model

In particular, monte carlo is designed after a “master/slave” model (maybe this is more commonly known as the “boss/worker” model). The master is in charge of creating slaves and tasking them to work. There may be any number of slaves distributed over a network. Master and slave communicate through sockets. Theoretically, a master and slave need not have the same file system. Each slave is responsible for requesting work, accomplishing work and reporting results. The work at hand is running a single simulation iteratively over an input space.

6.1.1 The Master

A master process tasks slaves to run the simulation with a given set of inputs. The master will task slaves to run in parallel. The master is responsible for keeping the slaves as busy as possible. To keep things running smoothly, the master is designed to reassign work when a slave is either dead or running too slowly. The master is only in charge of tasking work. The master does not run the simulation itself. The master will continue issuing work to the slaves until it is satisfied all simulation runs are complete.

6.1.2 Slaves

A slave consists of a parent and fork()ed children. A slave parent spawns a child using the fork() system call. A slave child runs the simulation in its own address space. Only one child exists at a time in a slave. Per slave, simulation execution is sequential.

A slave is responsible for requesting work from the master, running a Trick simulation with inputs given by the master, dumping recorded data to disk and informing the master when it is finished running its task.

6.2 Simulation Inputs

The goal of monte carlo is to run the simulation over a set of inputs. The inputs that the master passes to the slaves are either generated by a statistical algorithm or they are hard-coded by the user in a data file. Inputs may also be generated exclusively by user calculations.

6.3 Monte Carlo Output

For each simulation run within a monte carlo suite of runs, a directory called “MONTE_<name>” is created. Slave output is directed to this “MONTE_” directory. Trick recorded data is dumped to disk in a set of “RUN_” directories within the parent “MONTE_” directory. Along with recorded data, stdout and stderr are dumped. A file that contains the summary of all runs is dumped to the MONTE_ directory.

6.4 Data Processing

The trick_dp is designed to understand “MONTE_” directories. When choosing to plot a “MONTE_” directory, trick_dp will overlay all curves from each “RUN_” directory within the parent “MONTE_” directory. The plot widget has built in features that allow the developer to distinguish what curve goes with what simulation run.

6.5 Optimization

Optimization is made possible by creating a framework whereby the developer can change simulation inputs based on simulation results. Trick offers a set of job classes that allow the developer to enter the monte carlo loop and thereby enter the decision making on-the-fly. No canned optimization is available.

This special set of job classes work in concert together in master and slaves. Trick schedules jobs within the master at critical points so that they may create inputs to send to the slave as well as receive results from the slave. Slave jobs are scheduled to receive simulation inputs from the master as well as send simulation results back to the master.

The jobs are specified in the S_define. The jobs are created by the developer.

7.0 Conclusion

7.1 Communications

Throughout this document there have been boxes that elude to a communications package. While the design of this would be good for this document, it made sense to put it in the User's Guide. So if you are interested in the communications package, please refer to the User's Guide, there is an entire section dedicated to it.

7.2 Contacts

If you have any further questions about design, please contact Keith Vetter (vetter@titan.com), or Eddie Paddock (epaddock@titan.com).

8.0 Glossary

Automatic Code Generation	The process of generating compilable source code based upon a user defined specification.
context switching	The time it takes the operating system to load a new process for execution onto a CPU.
frame	An interval of time that encapsulates a set of jobs which have specific execution times.
hardware-in-the-loop	Term referring to a type of simulation that interfaces (output and/or feedback) with hardware (robotics, avionics, etc.) through an external I/O channel (VME, D/A, etc.) from the host computer to the hardware.
job	A simulation executive job; a source code module or subroutine.
itimers	A POSIX (not available on SUN4) interval timer that allows processes to establish system originated interval signals (SIGALRM) that can be handled by the process.
math model	A collection of source program subroutines which comprise a specific service or simulation building block; for example, earth environment, manipulator dynamics, Space Shuttle Orbiter control system, etc.
real-time	Term referring to the ability of simulation to guarantee execution rates at speeds identical to the system(s) they are simulating.
real-world clock	Real-time reference to actual time of day. Used to compare to simulation time when calculating overruns and under-runs.
runtime	Term referring to the time at which a simulation executes.
simulation	A collection of math models integrated, managed, and operated through the Trick Simulation Environment.
simulation developer	A person who builds simulations; must have good programming skills and in-depth knowledge of the capabilities and limitations of the Trick Simulation Environment.
simulation user	A person who operates a simulation; requires no programming skills, but does require an in-depth knowledge of the capabilities and limitations of the specific simulation being operated.
spawn	When a process (parent) starts another process (child) through UNIX functions (fork()/ exec()) or a system() call.
Trick	A simulation construction and operation environment.

9.0 Notes

None.

10.0 Appendices

None.

This Page Intentionally Blank

11.0 Index