AD-A244 977

(1)

SOFTWARE DESIGN DOCUMENT
CIG Host CSCI (9A)

DTIC
ELECTE
JAN 09 1992
S D
D

June, 1991

Prepared by:

BBN Systems and Technologies,
A Division of Bolt Beranek and Newman Inc.
10 Moulton Street
Cambridge, MA 02138
(617) 873-3000 FAX: (617) 873-4315

Prepared for:

Defense Advanced Research Projects Agency (DARPA)
Information and Science Technology Office
1400 Wilson Blvd., Arlington, VA 22209-2308
(202) 694-8232, AUTOVON 224-8232

Program Manager for Training Devices (PM TRADE)
12350 Research Parkway
Orlando, FL 32826-3276
(407) 380-4518

92-00261

92 1 6 064

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

SOFTWARE DESIGN DOCUMENT
CIG Host CSCI (9A)

June, 1991

Prepared by:

BBN Systems and Technologies,
A Division of Bolt Beranek and Newman Inc.
10 Moulton Street
Cambridge, MA 02138
(617) 873-3000 FAX: (617) 873-4315

Accesion For
NTIS
By
Dist
A-1

Prepared for:

Defense Advanced Research Projects Agency (DARPA)
Information and Science Technology Office
1400 Wilson Blvd., Arlington, VA 22209-2308
(202) 694-8232, AUTOVON 224-8232

Program Manager for Training Devices (PM TRADE)
12350 Research Parkway
Orlando, FL 32826-3276
(407) 380-4518

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

REPORT DOCUMENTATION PAGE

Form Approved OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE
June 1991

3. REPORT TYPE AND DATES COVERED
Software Design Document

4. TITLE AND SUBTITLE
Software Design Document CIG Host CSCI (9A)

5. FUNDING NUMBERS
Contract Numbers:
MDA972-89-C-0060
MDA972-89-C-0061

6. AUTHOR(S)
Author not specified.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Bolt Beranek and Newman, Inc. (BBN)
Systems and Technologies; Advanced Simulation
10 Moulton Street
Cambridge, MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER
Advanced Simulation #:
9112

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency (DARPA
3701 North Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
DARPA Report Number:
None.

11. SUPPLEMENTARY NOTES
None

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Statement A: Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
Distribution Code:
A

13. ABSTRACT *(Maximum 200 words)*

A Simulation Network (SIMNET) project Software Design Document that describes the Computer Image Generator (CIG) Host Computer Software Configuration Item (CSCI number 9A) of the SIMNET hardware and software training system for vehicle crew training and operational training.

14. SUBJECT TERMS
SIMNET Software Design Document for the CIG Host CSCI (CSCI 9A).

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Same as report.

Table of Contents

1 INTRODUCTION: CIG HOST CSCI

This document describes the 120TX/T Computer Image Generation (CIG) Host CSCI, also referred to as the CIG Real-Time Embedded code.

The CIG Host CSCI is the executable code that resides within the CIG and provides the Simulation Host (SIM) with an interface to the graphics hardware on the CIG.

1.1 The Simulator

A Vehicle Simulator Unit, or Simulator, consists of a CIG, a Simulation Host, one or more display monitors, a user, and the user's control mechanisms. Each Simulator simulates the actions of one combat vehicle, such as a tank, in real time. Multiple Simulators can be connected via a Simulation Network. The entire simulation exercise is controlled and coordinated by the Battle Manager using the Management, Command, and Control (MCC) system computer.

Once the MCC initializes a Simulator at the beginning of the exercise, the vehicle's crew directs the simulation. Each Simulator reports the position, orientation, and appearance of its simulated vehicle to the MCC and the other Simulators via the network.

Figure 1-1 illustrates the relationship between the CIG, the Simulation Host, and the MCC.

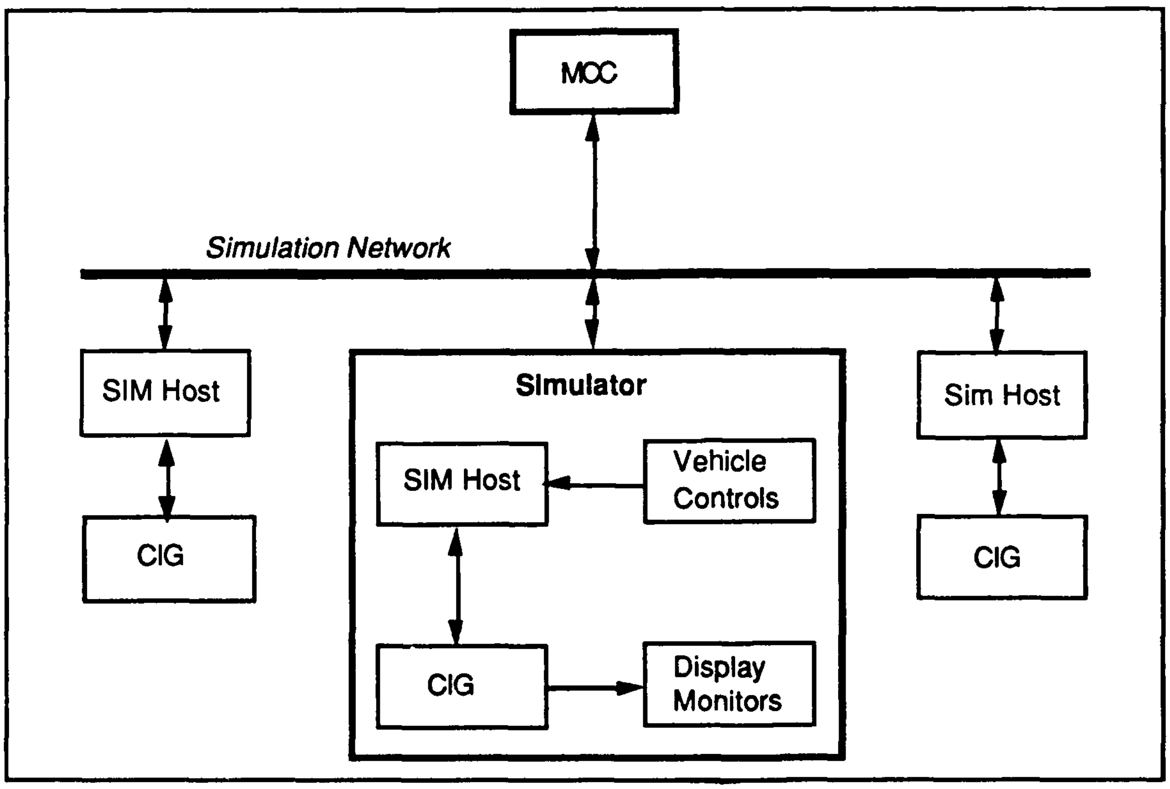


Figure 1-1. The Vehicle Simulator Unit (Simulator)

1.1.1 The Simulation Host

The Simulation Host receives and processes data from the simulation vehicle's mechanical controls, interfaces with the CIG, and communicates over the simulation network with other Simulators.

The Simulation Host is based on either a Masscomp or a Butterfly computer. The CIG's interface to the two is functionally the same, although some code modifications were required to interface to the Butterfly. These modifications do not affect the functionality of the CIG real-time software or the communication between the CIG and the Simulation Host. Code written specifically for the Butterfly platform is only cursorily addressed in this document.

1.1.2 The CIG

The CIG interfaces with the Simulation Host, controls the images in the simulation viewports (displays), and houses the database that describes the simulation terrain. The CIG is available in two models:

- The 120T CIG can generate up to eight low-resolution (320 by 200 pixels) views. These views are used in M1 and M2 Simulators.
- The 120TX CIG can generate one high-resolution (640 by 480 pixels) view or two low-resolution (320 by 240 pixels) views. These views are used in Stealth Simulators.

1.2 CIG-SIM Communication

The CIG and the Simulation Host communicate by exchanging 4K (4096-byte) message packets, each of which is a grouping of data messages. The physical interface to a Masscomp Simulation Host is a DR11-W communications device. The Butterfly platform uses a BVME interface.

Message packet exchanges occur every frame (every 66.7 milliseconds when running at 15 Hz). The CIG is the clock master for all synchronous message passing. Exchanges are initiated by the CIG after it detects a frame time event. Both the CIG and the Simulation Host have until the next frame to process information.

Message packets sent from the CIG describe the current state of the simulation vehicle. The Simulation Host uses this information to compute and update each parameter that affects the visual displays.

Message packets sent from the Simulation Host describe the new state of the simulation vehicle and/or changes to the simulation environment. Other messages specify where to display special effects, such as bomb blasts and smoke. The CIG uses this information to compute changes in the viewing displays.

The message structures used by the CIG and the Simulation Host to communicate are documented in the "120T/TX CIG-SIM Interface Manual."

1.3 CIG Software Structure

The CIG Host software is a multi-state, multi-tasking software system. It progresses through its various states upon receiving appropriate commands from the Simulation Host via the CIG-SIM message interface. The states of the CIG Host software are:

- Task Initialization
- System Configuration
- Real-Time Processing
- Stand-Alone (Flea) Mode

The simulation and other support software run as individual tasks. Using intertask mailbox locations, the tasks exchange information through shared memory. The tasks share system resources as needed, based on their relative priorities.

The top-level CSCs in the CIG Host CSCI are the following:

- Task Initialization (RTT)
- CIG Host Mainline (UPSTART)
  - Viewport Configuration
  - Data Traversal Processor (DTP) Command Generator
  - Real-Time Processing
  - 2-D Overlay Compiler [120TX systems only]
- Database Management (ROWCOL_RD)
- Database Feedback (LOCAL_TERRAIN)
- Ballistics Processing (BALLISTICS)
- User's Interface (GOSSIP)
- Stand-Alone Message Interface (FLEA)
- Force Processor Task (FORCETASK) [120TX systems only]

Figure 1-2 illustrates these CSCs.

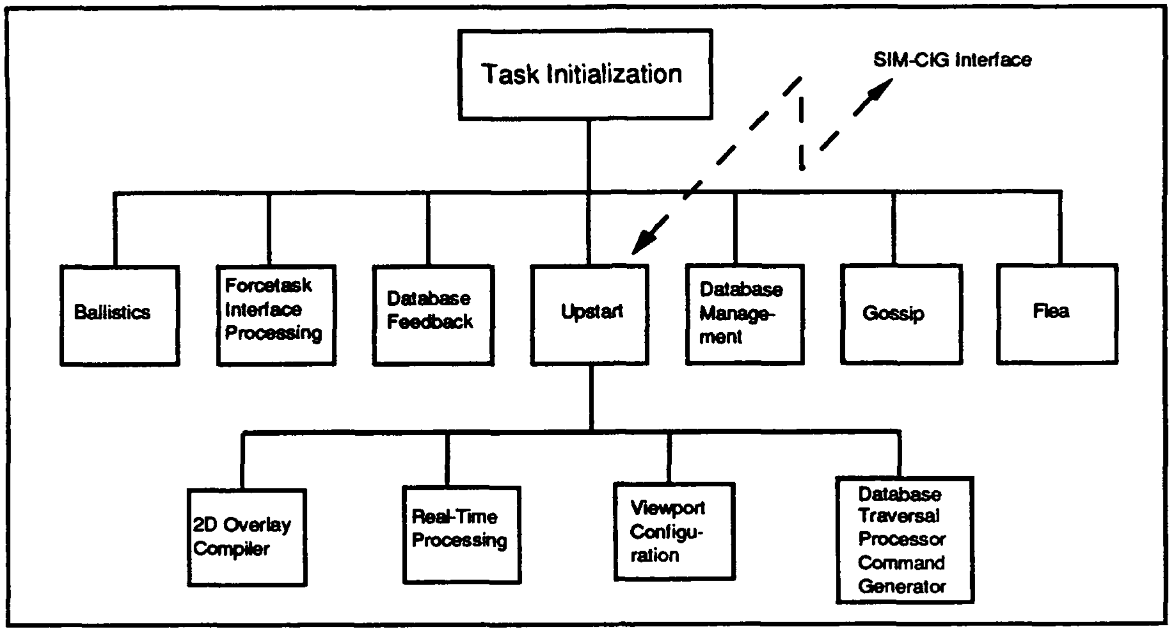


Figure 1-2. CIG Embedded Software CSCs

1.4 How This Document Is Organized

Section 1 (Introduction)
Provides a general overview of the CIG Embedded Software, the Simulation Host, and the Vehicle Simulator Unit.

Section 2 (CSC Descriptions)
Describes each CSC in the CIG Embedded Software CSCI. Each subsection begins with a general overview of the CSC, its major data structures, the primary functions it performs, and how it relates to the other CSCs. This is followed by a detailed description of each CSU in the CSC. The CSUs are presented in alphabetical order.

For the purposes of this document, a CSU is defined as a source code (.c or .asm) file. CSUs are documented as follows:

- The section heading identifies the name of the source file.
- If a CSU contains multiple functions, each is described in a separate subsection under the CSU section heading. The functions are described in the order in which they appear in the source file.
- If a CSU contains only one function, it is described under the CSU section heading. If the function name differs from the CSU name, the function name is shown in parentheses following the CSU name. If the function name matches the CSU name (minus the .c or .asm suffix), the function name is not shown in the heading.

The description of a function includes its general purpose, its function call, definitions of its parameters and return values, and a description of its processing. The description also identifies all called and calling routines.

Section 3 (Resource Utilization)
Provides disk and memory usage statistics.

Appendix A (System Include Files)
Describes the contents of each header (.h) file used in the system, and identifies the CSUs that include it. All include files are listed in alphabetical order.

Appendix B (System Macros)
Describes the macros used to perform specialized functions throughout the system, and identifies where they are used. All macros are listed in alphabetical order.

Appendix C (Operating System Service Calls)
Briefly describes the operating system service libraries and standard C libraries used by the CIG functions.

Appendix D (Glossary Of Terms And Abbreviations)
Defines some of the specialized terminology, abbreviations, and acronyms used in this document.

Appendix E (Cross-Reference Tables)
Provides lists that may help the reader locate CSUs, data type definitions, functions, and macros.

2 CSC DESCRIPTIONS

The CSCs that make up the CIG Host software system are the following:

Task Initialization (RTT)
Initiates the execution of the other CIG Host tasks.

CIG Host Mainline (UPSTART)
Configures the viewport displays, generates DTP commands, runs the real-time simulation, and generates two-dimensional overlays.

Database Management (ROWCOL_RD)
Reads new rows or columns of load modules from the terrain database into active area memory as required.

Database Feedback (LOCAL_TERRAIN)
Sends information describing the local terrain (the area around the simulated vehicle) to the Simulation Host, based on the simulated vehicle's current position.

Ballistics Processing (BALLISTICS)
Determines which load modules and grids in the database are intersected by a given chord.

User Interface (GOSSIP)
Provides a back-door user interface that allows certain debugging and query features during runtime operation.

Stand-Alone Host Emulator (FLEA)
Emulates the Simulation Host for stand-alone CIG operation and testing.

Force Processor (FORCE)
On the 120TX CIG only, controls the interface between the CIG real-time task and the two-dimensional overlay processor task.

This section describes the functions performed by each of these CSCs.

2.1 Task Initialization (RTT) CSC

This section details the software that performs the task initialization phase of the CIG Host system. The task initialization CSU, rtt, is the initial task in the CIG Real-Time Software. It is executed from the user's terminal or via the auto-boot mechanism. rtt initiates the execution of all other tasks in the CIG Host CSCI, then terminates itself.

As shown in Figure 2-1, this CSC contains only one CSU: rtt.c. The functions in rtt.c are described in this section.

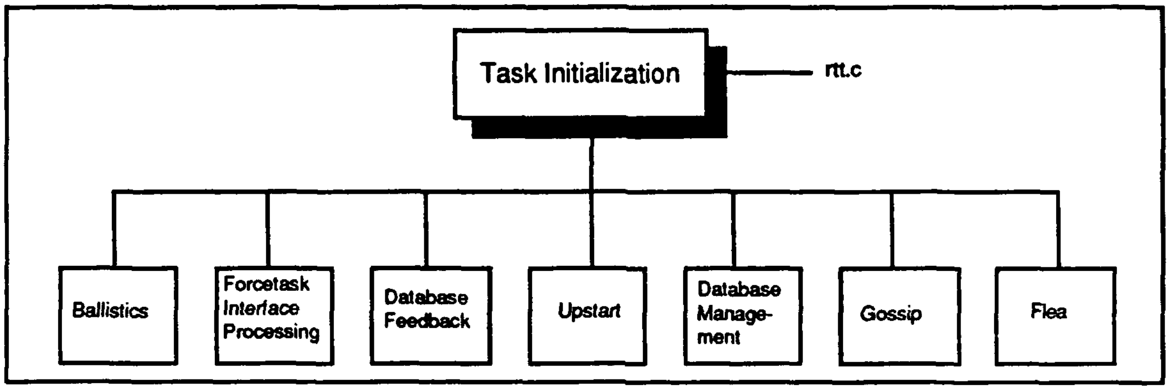


Figure 2-1. Task Initialization CSU

2.1.1 rtt.c

The rtt.c CSU contains the functions responsible for task and queue initialization. These functions are:

- apinit
- qassign
- tassign

2.1.1.1 apinit

The apinit function is a high-priority task created by the system. apinit creates all application queues and tasks, runs all tasks, and then deletes itself from the system.

The function call is **apinit()**. apinit does the following:

- Calls bus_error to determine which type of Ballistics board is in the CIG.
- Adds a 45-second system delay for the lamplighter if switch 5 is on ("go flying" mode) and switch 1 is off (auto-boot mode).
- Initializes the application task id and queue id.
- Inserts the application task table into the system task table.
- Calls tassign to assign a task id to each task.
- Calls qassign to assign a queue id to each queue.
- Deletes its own task from the system.

apinit initiates the application task table in the operating system by establishing entries for the other CSCs in the real-time software, as follows:

name	tid	priority	type	queue	qsize	entry
"upstart"	yes	2	task	no	0	upstart
"flea"	yes	10	task	yes	16	flea
"local_terrain"	yes	8	task	no	0	local_terrain
"ballistics"	yes	6	task	no	0	bx_task
"rowcol_rd"	yes	4	task	no	0	rowcol_rd
"gossip"	yes	12	task	no	0	gossip

Called By: none

Routines Called: bus_error
printf
qassign
rotate_x_nt
rotate_y_nt
rotate_z_nt
sc_tdelete
strcpy
tassign
translate

Parameters: none

Returns: none

2.1.1.2 qassign

The qassign function assigns and creates all queues. The only task for which a queue is created is flea, with a queue size of 16.

The function call is **qassign(qsize)**, where *qsize* is the size of the queue to be created. qassign does the following:

- Increments the queue identifier number by 1.
- Verifies that the queue size is valid.
- Calls sc_qcreate to create the queue.
- Returns the queue id (*apqid*) to apinit.

The function returns -1 if the queue size is specified as 0 or "no."

Called By:	apinit	
Routines Called:	sc_qcreate	
Parameters: .	int	qsize
Returns:	-1 apqid	

2.1.1.3 tassign

The tassign function assigns and creates the upstart, rowcol_rd, ballistics, local_terrain, flea, and gossip tasks.

The function call is **tassign(tflag, tentry, tpri)**, where:

tflag is "yes" (identifying this as a task)
tentry is the task's entry point (name)
tpri is the task's priority

tassign does the following:

- Increments the task identifier number by 1.
- Verifies that *tflag* is not "no."
- Calls sc_tcreate to create the task.
- Returns the task id (*aptid*) to apinit.

The function returns -1 if *tflag* is "no."

Called By:	apinit	
Routines Called:	sc_tcreate	
Parameters:	int char int	tflag *tentry tpri
Returns:	-1 aptid	

2.2 CIG Host Mainline (UPSTART) CSC

The CIG Host Mainline CSC, UPSTART, contains the functions responsible for configuring the viewports (simulator displays) and running the simulation.

The Simulation Host controls all functions of the visual simulation and determines what information is sent to the CIG. The CIG uses this information to control the images in the viewports of the visual simulator.

Upon request from the Simulation Host, the simulation goes into database setup mode, where memory is initialized and the appropriate database subsection is loaded into active area memory (AAM). From setup mode, the Simulation Host can request a transition to simulation mode. This causes a local terrain message request, enables system frame interrupts, and initializes system variables.

Every frame, the simulation does the following:

- Waits for the system interrupt.
- Prepares a laser range message.
- Sends a message packet to the Simulation Host.
- Receives a message packet from the Simulation Host.
- Determines which buffer in double-buffer memory to use. (Double buffering allows one buffer to be used by the hardware while the other is being updated by the software. The simulation and the hardware switch buffers on every exchange, so the hardware is always accessing the most recently updated information.)
- Restores the model return addresses.
- Processes one "My vehicle" message which:
    - Expands the eight matrices (one per viewport) of the simulation vehicle.
    - Loads 11 overlay characters into the gunner channel.
    - Tells the T&C (Timing and Control) board which channels to display.
- Processes zero or more "other vehicle" messages, each of which:
    - Expands one to three matrices for vehicles in the terrain.
    - Adds a model to the proper load module.
    - Displays smoke and fire if appropriate.
- Processes zero or more "show effect" messages, each of which:
    - Stores effect data.
    - Adds an effect to the proper load module.
- Processes zero or one trajectory chords.
- Reprocesses zero or more "show effect" messages from previous frames.

Every 32 frames, the simulation constructs and sends a message on the contents of the local terrain. This message contains data regarding the terrain, roads, rivers, and buildings that lie in the four grids surrounding the simulated vehicle. This information is used by the Simulation Host to provide collision detection with objects in the simulated environment, and to calculate the correct vehicle dynamics for driving on the terrain.

When complete, the Simulation Host may stop the simulation to enable going into another mode, or may reconfigure the Simulator in another area.

The major functional components of UPSTART are as follows:

Viewport Configuration
Initializes and builds the viewport configuration tree before runtime. The configuration tree describes the relationship between each physical component of the simulated vehicle and the location of the viewports.

DTP Command Generator
Generates data traversal processor (DTP) hardware commands from the viewport configuration tree.

Real-Time Processing
Runs the simulation using messages passed between the Simulation Host and Ballistics.

2-D Overlay Compiler
Builds the 2-D (two-dimensional) overlays, and generates executable commands for the 2-D processor on 120TX CIGs.

Figure 2-2 illustrates the components of the UPSTART CSC. The following subsections describe the CSUs in each of these functional areas, in the order listed above.

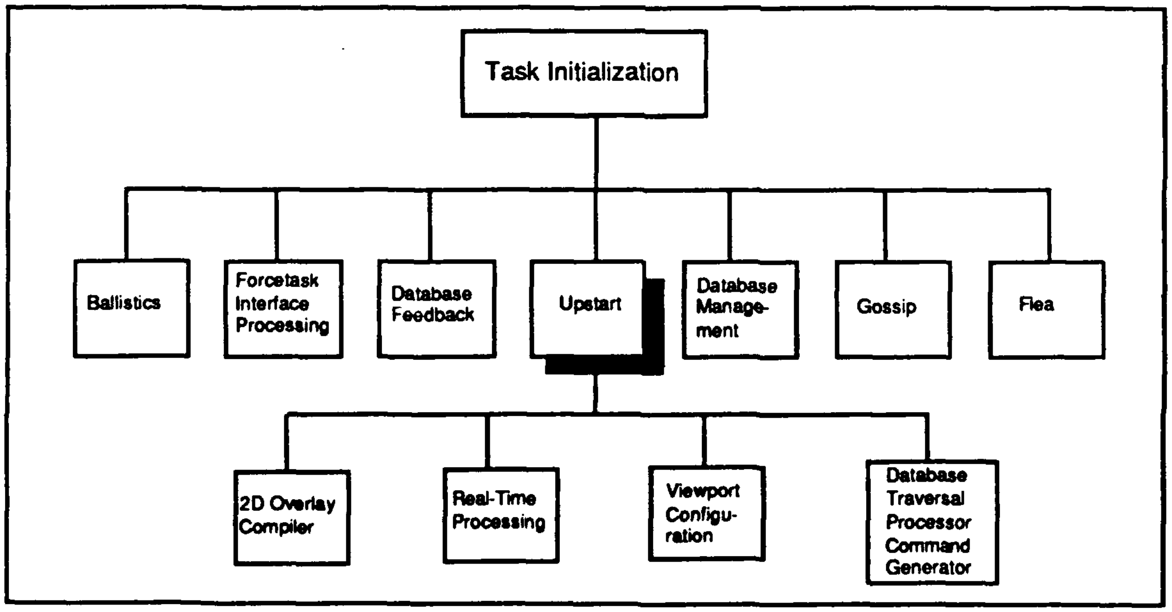


Figure 2-2. UPSTART Functional Components

2.2.1 Viewport Configuration

Viewport Configuration is the area of UPSTART that is responsible for initializing and building the configuration tree before runtime. The configuration tree describes the relationship between each physical component of the simulated vehicle and the location of the viewports. The messages used to set up the configuration tree are received from the Simulation Host.

The configuration tree consists of the following:

- One root node, which marks the start of the configuration tree. This node contains no data and must be the first node created.

- One or more matrix nodes, each of which contains a transformation matrix that specifies rotation angles (heading, pitch, and roll) and translation values. The matrices in all nodes in a traversal path of the tree are concatenated to generate the view of the world for the viewport represented by that path. Matrix nodes are designated as either dynamic (ones that are updated during the simulation) or static (ones that do not change during the simulation).

- Zero or more conditional (branch) nodes, each of which branch into one of two traversal paths based on a runtime condition. The node branched to if the condition is true is the conditional node's "true child" and the node branched to if the condition is false is the "false child." The branch values are stored in the system view flags array. The branch values in effect at any given time in the simulation are set via messages sent from the Simulation Host.

- Viewport parameters for each viewport. These parameters are the screen resolution, viewing range, near plane, field-of-view angles, level-of-detail multiplier, and aspect ratio (currently not used). Viewport parameters are associated with the final node in each traversal path in the configuration tree.

  Note that the same viewport may be defined multiple times, each with different parameters. A conditional node enables a change to new viewport parameters during the simulation.

- One or more sets of graphics path parameters for each viewport. A graphics path is a window on a viewport. On the 120T, there is one graphics path per viewport. On the 120TX, there may be two or four, depending on the resolution. The graphics path parameters are used to load the hardware.

The structure of the configuration tree cannot be changed during runtime — all nodes and viewport definitions must be created at CIG initialization time. However, various parameters *within* the configuration tree do change during the simulation. Therefore, some Viewport Configuration functions are called by simulation (in the Real-Time Processing component) to update configuration tree structures during runtime.

Specifically, messages can be used to update the following structures after the configuration tree has been created:

- Dynamic matrices. The Simulation Host can provide a new matrix or a change (e.g., rotation) to the current matrix.

- Branch values for the conditional nodes. Changing the branch values during a simulation causes selection of a different traversal path and, usually, different viewport parameters.
- Certain viewport parameters (the level-of-detail multiplier and the field-of-view angles). Although a message is available to change these parameters directly, it is recommended that all desirable viewport parameter combinations be built into the configuration tree and selected using branch values.

The configuration tree can contain a maximum of 64 nodes. Every node is referenced by a unique index, which is used in messages sent to update the node during the simulation. The root node is always assigned node index 0. A node that has viewport parameters attached to it must have a node index between 1 and 31.

Every matrix node in the configuration tree must be defined in one of two formats: RTS4x3 (4 x 3 rotation translation scale) or HPRXYZS (3 x 3 scale heading pitch roll translation). A matrix node's format can be redefined during the simulation.

The format of each of these matrix structures is as follows:

RTS4x3 (4 x 3 rotation translation scale)

The matrix format is:

rotation[0,0]	rotation[0,1]	rotation[0,2]
rotation[1,0]	rotation[1,1]	rotation[1,2]
rotation[2,0]	rotation[2,1]	rotation[2,2]
translation.x	translation.y	translation.z

where:

rotation is an angle in degrees
translation is a distance in meters

The typedef for this matrix structure is:

```
typedef struct {
REAL_4      rotation[3][3];
R4P3D       translation;
} RTS4x3_MTX;
```

HPRXYZS (3 x 3 scale heading pitch roll translation)

The matrix format is:

heading	pitch	roll
translation.x	translation.y	translation.z
scale.x	scale.y	scale.z
scale order	heading order	pitch order
roll order	translate order	

where:

heading = -yaw = -z rotation in degrees
pitch = x rotation in degrees
roll = y rotation in degrees
translation is a distance in meters

scale is a scaling factor (used to enlarge or reduce matrices)
order values specify the order in which the matrices are to be concatenated

The typedef for this matrix structure is:

```
typedef struct {
REAL_4       heading;
REAL_4       pitch;
REAL_4       roll;
R4P3D        translation;
R4P3D        scale;
BYTE         concat_order[5];
} RTS3x3_MTX;
```

A third matrix format, **ROT2x1 (2 x 1 rotation)**, can be used to rotate a matrix along one axis. Matrix nodes cannot be defined as this matrix format, although they can be updated by it. The matrix format for ROT2x1 is:

$$\begin{bmatrix} \cos(\text{rotation}^{o}) & \sin(\text{rotation}^{o}) \\ \text{rotation axis} & \end{bmatrix}$$

where:
rotation is the angle of rotation in degrees
rotation axis is the axis along which rotation is to occur: 0 (x), 1 (y), or 2 (z)

The typedef for this matrix structure is:

```
typedef struct {
REAL_4       cos_rotation;
REAL_4       sin_rotation;
BYTE         rotation_axis;
} ROT2x1_MTX;
```

The functions in Viewport Configuration do the following:

- Create all configuration nodes, viewport parameter entries, and graphics path entries, based on data received from the Simulation Host.
- Generate DTP-style matrices from the matrices provided by the Simulation Host.
- Set up calibration, gunner, and gun barrel overlays for 120T systems. (These are hard-coded overlays that can be displayed on a viewport on top of the terrain display.)
- Generate DTP code for the overlays.
- Process the system view flags/branch values and load them into the T&C (Timing and Control) board.

Usually, the configuration tree is built according to messages received from the Simulation Host. To initiate this process, db_mcc_setup (in the Real-Time Processing component) calls the cig_config function. cig_config in turn calls other Viewport Configuration functions to allocate memory and configure the nodes, viewports, and view flags.

A configuration tree can also be created from data in an ASCII file that is created off-line and installed on the CIG. The read_configfile function is used to parse this file and call the appropriate functions to create the tree. This method is provided for stand-alone use and testing.

Figure 2-3 identifies the CSUs in Viewport Configuration. The functions performed by these CSUs are described in this section.

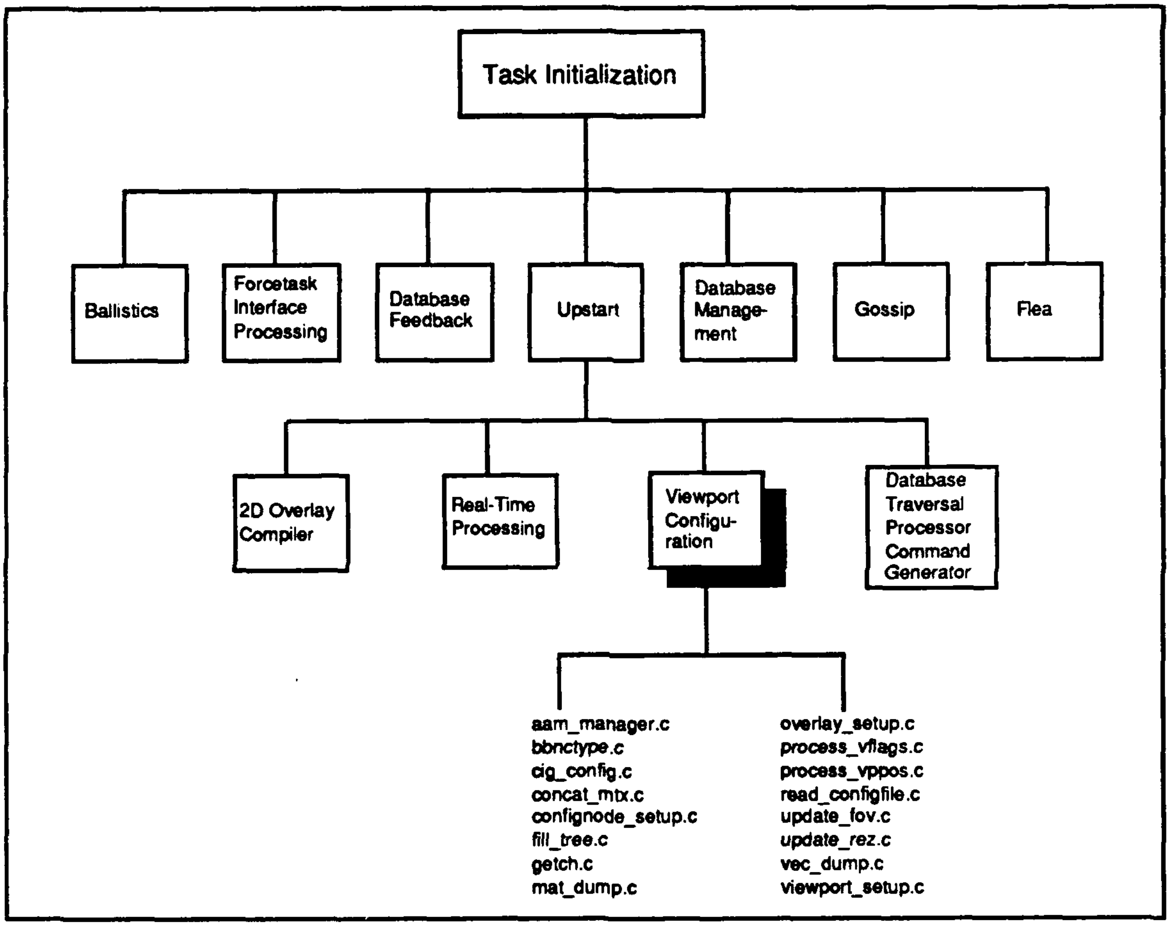


Figure 2-3. Viewport Configuration CSUs

Figure 2-4 illustrates how the major functions of Viewport Configuration interact with each other to create the configuration tree based on messages received from the Simulation Host.

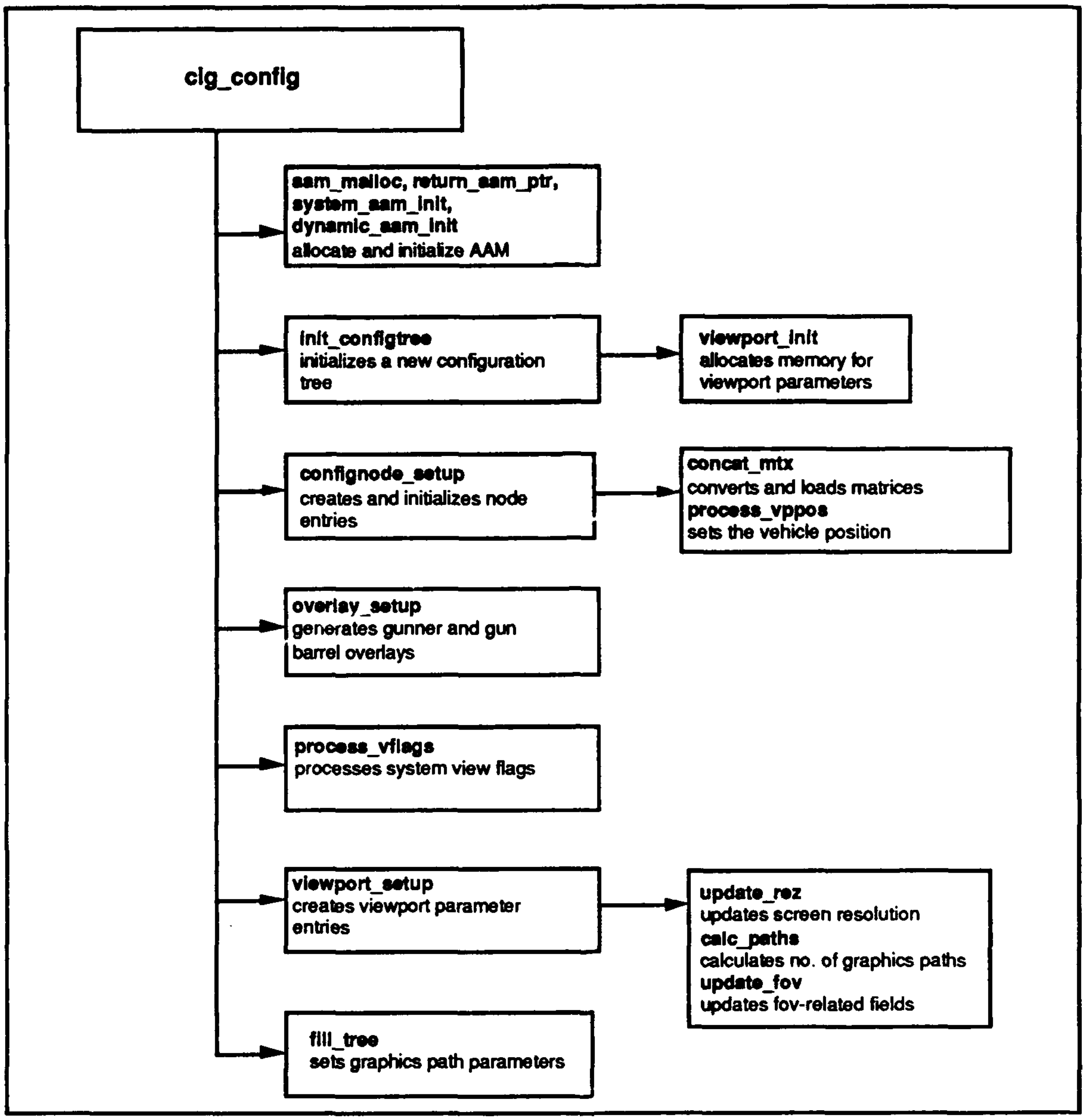


Figure 2-4. Viewport Configuration Flow Diagram

2.2.1.1 aam_manager.c

The functions in aam_manager.c are used to allocate and manage the system (static) and dynamic areas of active area memory. Dynamic memory is located in the double-buffer area; static memory is not double-buffered.

The functions in aam_manager.c are:

- aam_malloc
- return_aam_ptr
- system_aam_init
- dynamic_aam_init

2.2.1.1.1 aam_malloc

The aam_malloc function allocates system and dynamic memory.

The function call is **aam_malloc(static_flag, num_of_bytes)**, where:

static_flag identifies the area of memory (SYSTEM or DYNAMIC)
num_of_bytes is the number of bytes of memory requested

When it receives a request to allocate active area memory, aam_malloc does the following:

- Determines which area of memory is being requested.
- Verifies that sufficient memory is available.
- Allocates the memory and returns a pointer (*temp_ptr*) to it.

If there is insufficient memory to process the request, aam_malloc returns NULL and displays the amount of memory available.

Called By:	cig_config confignode_setup init_configtree viewport_setup	
Routines Called:	printf	
Parameters:	BYTE WORD	static_flag num_of_bytes
Returns:	temp_ptr NULL	

2.2.1.1.2 return_aam_ptr

The return_aam_ptr function returns the address of the next available location in the static or dynamic area of active area memory.

The function call is **return_aam_ptr(static_flag)**, where *static_flag* identifies the area of memory (SYSTEM or DYNAMIC).

return_aam_ptr returns *system_aam* (the next available address in static memory) or *dynamic_aam* (the next available address in dynamic memory).

Called By:	cig_config
Routines Called:	none

Parameters:	BYTE	static_flag
Returns:	system_aam dynamic_aam	

2.2.1.1.3 system_aam_init

The system_aam_init function initializes the system (static) section of active area memory.

The function call is **system_aam_init(system_aam_add, limit)**, where:

system_aam_add is the starting address of the memory to be initialized
limit is the ending address of the memory to be initialized

The function returns *system_aam*, the starting address of the initialized memory.

Called By:	cig_config	
Routines Called:	none	
Parameters:	WORD WORD	system_aam_add limit
Returns:	system_aam	

2.2.1.1.4 dynamic_aam_init

The dynamic_aam_init function initializes the dynamic section of active area memory.

The function call is **dynamic_aam_init(dynamic_aam_add, limit)**, where:

dynamic_aam_add is the starting address of the memory to be initialized
limit is the ending address of the memory to be initialized

The function returns *dynamic_aam*, the starting address of the initialized memory.

Called By:	cig_config	
Routines Called:	none	
Parameters:	WORD WORD	dynamic_aam_add limit

Returns: dynamic_aam

2.2.1.2 bbnctype.c

bbnctype is a runtime library that defines control characters, punctuation, digits, and alphas. This file is not currently used.

Called By: none

Routines Called: none

Parameters: none

Returns: none

2.2.1.3 cig_config.c

The functions in the cig_config.c CSU initialize and manage the configuration tree. These functions are:

- cig_config
- init_configtree
- free_configtree

2.2.1.3.1 cig_config

The cig_config function is the CIG configuration message handler. It is responsible for setting up the configuration tree before runtime. cig_config is called by db_mcc_setup (in the Real-Time Processing component of UPSTART) when the CIG Control message from the Simulation Host specifies C_CIG_CONFIG.

The function call is **cig_config(state)**, where *state* is the current state of the CIG system (C_CIG_CONFIG). cig_config does the following:

- Calls system_aam_init to initialize and set up a pointer to the system section of active area memory.
- Calls dynamic_aam_init to initialize and set up a pointer to the dynamic section of active area memory.
- Calls init_configtree to initialize a new configuration tree and get pointers to the tree and its associated structures.
- Calls aam_malloc to allocate 16 view mode words and the daylight TV thermal word (*dtv_therm_word*).
- Initializes the calibration modifier.
- Loads the reconfiguration data that goes into double-buffered active area memory into DB0.

- Calls make_cal_overlay to create the calibration overlay.
- Initializes *agl_wanted* to false. This flag can be set true by the Simulation Host to enable AGL (above ground level) processing. If AGL processing is enabled via the MSG_AGL_SETUP message, the simulated vehicle's altitude above ground level is calculated and returned to the Simulation Host every frame.
- Processes each configuration message received from the Simulation Host in turn (see table below).
- When a CIG Control-Stop message is received, returns a pointer to the top of the newly created configuration tree to db_mcc_setup.

The following table summarizes the processing performed by cig_config in response to each valid message type it receives from the Simulation Host. The first column lists the messages in alphabetical order. The second column briefly describes the purpose of the message (in italics), then lists the major steps performed by cig_config to process the message.

Message from SIM Host	Processing by cig_config
MSG_AGL_SETUP	*Toggles AGL processing on/off.* Sets agl_wanted in global memory.
MSG_AMMO_DEFINE	*Define ammunition maps.* Sets ammo_map in global memory.
MSG_CIG_CTL C_NULL C_STOP	*Causes a transition to another performance state.* No action. Calls fill_tree; calls dtp_compiler; copies reconfigurable viewport data from DB0 to DB1; returns a pointer to the top of the configuration tree to db_mcc_setup.
MSG_CREATE_CONFIGNODE	*Creates a configuration tree node entry.* Calls confignode_setup.
MSG_DR11_PKT_SIZE	*Specifies exchange packet parameters.* Sets CIG and SIM exchange packet size, local terrain chunk size, and local terrain message interval.
MSG_END	*Signals end of packet buffer.* Calls EXCHANGE_DATA to send output and receive input buffers.
MSG_GEN_CONFIGTREE	*Not currently implemented.*
MSG_OVERLAY_SETUP	*Places overlays on specified viewports.* Calls overlay_setup.
MSG_VIEW_FLAGS	*Sets system view flags (on/off, daylight/TV, etc.).* Calls process_vflags.
MSG_VIEWPORT_STATE	*Defines all viewport parameters.* Calls viewport_setup.

Called By: db_mcc_setup

Routines Called: aam_malloc
confignode_setup
dtp_compiler
dynamic_aam_init

EXCHANGE_DATA
fill_tree
init_configtree
make_cal_overlay
overlay_setup
printf
process_vflags
read
return_aam_ptr
sc_pend
sc_post
SYSERR
system_aam_init
viewport_setup
write

Parameters:	INT_2	state
Returns:	top_of_configtree	

2.2.1.3.2 init_configtree

The init_configtree function initializes memory and pointers for the configuration tree. This function is called by cig_config before it begins processing messages from the Simulation Host.

The function call is **init_configtree(n_nodes, n_views, n_paths)**, where:

n_nodes is the number of configuration nodes in the tree
n_views is the number of viewport parameter entries in the tree
n_paths is the number of graphics path entries in the tree

init_configtree does the following:

- Allocates memory for the configuration tree, for the number of nodes requested.
- Allocates memory for the viewport positions array and stores a pointer to it in *child_ptr[1]* of the root configuration node. The viewport positions (vppos) array stores the current location of the simulation vehicle.
- Sets up an array for the system view flags and branch values, and stores a pointer to it in the *branch_value* pointer of the root configuration node.
- Allocates memory for the viewport parameters, based on the number of entries requested.
- Calls viewport_init to initialize the viewport parameter variables.
- Allocates memory for the graphics path parameters, based on the number of entries requested.

The function returns 1 if the configuration tree was initialized successfully. It returns 0 if memory could not be allocated for the tree or for any of the structures.

Called By: cig_config

Routines Called:	aam_malloc calloc viewport_init	
Parameters:	WORD WORD WORD	n_nodes n_views n_paths
Returns:	1 (SUCCEED) 0 (FAIL)	

2.2.1.3.3 free_configtree

The free_configtree function deallocates memory and pointers for the configuration tree, including the viewport and graphics path structures. This function is called by db_mcc_setup (in the Real-Time Processing component) after a real-time simulation has ended.

The function call is **free_configtree()**.

Called By:	db_mcc_setup
Routines Called:	free
Parameters:	none
Returns:	none

2.2.1.4 concat_mtx.c

The concat_mtx function generates DTP-style matrices from the matrices provided by the Simulation Host, and loads the matrices into active area memory. This function is called by confignode_setup to generate and load the initial matrix for each matrix node during viewport configuration. It is called by simulation to update dynamic matrices during runtime if any of the following messages is received from the Simulation Host: MSG_ROT2x1_MATRIX, MSG_RTS4x3_MATRIX, MSG_HPRXYZS_MATRIX, MSG_TRANSLATION, MSG_SCALE, MSG_1ROTATION, or MSG_3ROTATIONS.

The function call is **concat_mtx(config_node, matrix, db)**, where:

config_node is a pointer to the configuration node
matrix is the original matrix
db is the double-buffer memory current base pointer

concat_mtx does the following:

- Determines the Simulation Host matrix type (RTS4x3, ROT2x1, or RTS3x3).
- Unpacks the Simulation Host matrix.
- For an RTS4x3 matrix:
  - Calls mtxcpy to copy the new matrix.
- For an ROT2x1 matrix:
  - Determines which axis the matrix is to be rotated along.
  - Updates the matrix's rotation values.
- For an RTS3x3 (HPRXYZS) matrix:
  - Calls id_4x3mtx to create an identity matrix.
  - Determines the concatenation order specified in the message.
  - Performs the concatenation in the specified order:
    scale - Calls id_4x3mtx, calls scale_mtx, calls getmatrix.
    heading - Calls id_4x3mtx, calculates *cos_theta* and *sin_theta*, calls rotate_z_nt, calls getmatrix.
    pitch - Calls id_4x3mtx, calculates *cos_theta* and *sin_theta*, calls rotate_x_nt, calls getmatrix.
    roll - Calls id_4x3mtx, calculates *cos_theta* and *sin_theta*, calls rotate_y_nt, calls getmatrix.
    translate - Calls id_4x3mtx, calls translate, calls getmatrix.
- Calls mtxcpy to load the new or modified matrix into active area memory.

If an error is detected, concat_mtx sets *err_code* to TRUE.

Called By:	confignode_setup simulation	
Routines Called:	getmatrix id_4x3mtx mtxcpy mult_4x3mtx r4mat_dump (in debug mode only) rotate_x_nt rotate_y_nt rotate_z_nt scale_mtx translate	
Parameters:	CONFIGURATION_NODE MTXUNION INT_4	*config_node matrix db
Returns:	err_code	

2.2.1.5 confignode_setup.c

The confignode_setup function creates and initializes node entries in the configuration tree. confignode_setup is called by cig_config if the message from the Simulation Host is MSG_CREATE_CONFIGNODE.

The function call is **confignode_setup(imsg, top_of_configtree, viewport_params, path_params, db)**, where:

imsg is a pointer to the message (MSG_CREATE_CONFIGNODE)
top_of_configtree is a pointer to the configuration tree's root node
viewport_params is a pointer to the viewport parameters
path_params is a pointer to the graphics path parameters
db is the double-buffer memory current base pointer

confignode_setup does the following:

- Sets up all configuration tree-related pointers.
- If configuring the root node:
  - Resets the vehicle id to 0.
- Initializes the parent index to an invalid value (-1).
- Loads the parent pointer into the configuration tree node.
- If configuring a child of a conditional node:
  - For the false child, loads a pointer to it in the parent's false pointer slot.
  - For the true child, loads a pointer to it in the parent's true pointer slot.
- If configuring a child of a matrix node:
  - For an only child, load a pointer to it in the parent's first pointer slot.
  - For a child with siblings, sets the youngest sibling's pointer to the new node.
- If configuring a matrix node:
  - Generates the matrix.
  - Loads the matrix into active area memory.
- If configuring a conditional node:
  - Sets the branch value pointer using the Simulation Host index into the branch value array. (The address of this array is in the root node's branch value pointer.)
- If configuring a word/hull matrix node (i.e., a child of the root node):
  - Sets the vehicle id.
  - Loads the corresponding viewport position into the view positions (vppos) array.

Called By:	cig_config read_configfile
Routines Called:	aam_malloc concat_mtx mtxcpy process_vppos strcpy

Parameters:	WORD	*imsg
	CONFIGURATION_NODE	*top_of_configtree
	VIEWPORT_PARAMETERS	*viewport_params
	GRAPHICS_PATH_PARAMETERS	*path_params
	INT_4	db
Returns:	none	

2.2.1.6 fill_tree.c

The fill_tree.c CSU contains two functions:

- fill_tree
- power

2.2.1.6.1 fill_tree

The fill_tree function sets the graphics path flags in configuration tree nodes. fill_tree is called by cig_config when the message from the Simulation Host is C_STOP, indicating that all configuration node messages have been sent.

The function call is **fill_tree(graphics_path)**, where *graphics_path* is a pointer to the graphics path parameters.

fill_tree does the following:

- Uses the graphics path entry path id to set a bit in the configuration node path flag. For example, if the path id is 4, the path flag is set to 0001 0000.
- Traverses up the configuration tree, setting the path flags in the configuration nodes.

Called By:	cig_config read_configfile	
Routines Called:	power	
Parameters:	GRAPHICS_PATH_PARAMETERS	*graphics_path
Returns:	none	

2.2.1.6.2 power

The power function raises a base to a power. This function is called by fill_tree when it traverses the configuration tree.

The function call is **power(base, n)**, where:

base is the base to be raised
n is the power

The calculated value is returned as *result.*

Called By:	fill_tree	
Routines Called:	none	
Parameters:	WORD WORD	base n
Returns:	result	

2.2.1.7 getch.c

The getch function gets a character from a configuration file and returns it as *ch.*

The function call is **getch(fdi)**, where *fdi* is a unique identifier associated with the file.

Called By:	read_configfile REAL4_fscanf STRING_fscanf WORD_fscanf	
Routines Called:	cmd	
Parameters:	INT	fdi
Returns:	ch	

2.2.1.8 mat_dump.c

The functions in mat_dump.c are used to dump matrices to the standard output (stdout). These functions are:

- r4mat_dump
- r8mat_dump

2.2.1.8.1 r4mat_dump

The r4mat_dump function dumps a matrix to stdout. This function is called only if debug mode is enabled.

The function call is **r4mat_dump(str, mat)**, where:

str is a string to display (on stdout) to describe the matrix
mat is a pointer to the area of active memory that contains the matrix

Called By:	concat_mtx	(in debug mode only)
	viewspace_mtx	(in debug mode only)
Routines Called:	printf	
Parameters:	char	*str
	REAL_4	mat[3][3]
Returns:	none	

2.2.1.8.2 r8mat_dump

The r8mat_dump function dumps a matrix to stdout.

The function call is **r8mat_dump(str, mat)**, where:

str is a string to display (on stdout) to describe the matrix
mat is a pointer to the area of active memory that contains the matrix

This function is not currently used.

Called By:	none	
Routines Called:	printf	
Parameters:	char	*str
	REAL_8	mat[3][3]
Returns:	none	

2.2.1.9 overlay_setup.c

The overlay_setup function is a message handler that sets up calibration, M1 and M2 gunner overlays, and M1 and M2 gun barrel overlays. It also generates DTP code for the overlays. overlay_setup is called by cig_config when the message from the Simulation Host is MSG_OVERLAY_SETUP.

The function call is **overlay_setup(pmsg, pview)**, where:

pmsg is a pointer to the MSG_OVERLAY_SETUP message
pview is a pointer to the viewport parameters

overlay_setup does the following:

- Calls make_m1_overlays or make_m2_overlays to create the gunner and gun barrel overlays.
- Inserts the gun barrel data into the viewport parameter nodes.

Overlays are hard-coded displays of three-dimensional polygons that are displayed on a viewport, super-imposed over the view of the terrain. The overlay shows non-terrain objects that would normally be seen when looking outside the vehicle's window. For example, gun overlays show those parts of the simulated vehicle that would be visible from the window, obscuring the view of the terrain. Gunner overlays show cross-hairs and numerical readouts of simulation parameters.

Any node that has viewport parameters and has bit 0 of the node's branch mask set has the gunner's overlay placed on the viewport. Similarly, any node that has viewport parameters and has bit 1 of the node's branch mask set has the gun barrel added to its processing.

Gunner, gun barrel, and calibration overlays are used by the 120T CIG only. Overlays on the 120TX are generated through the 2-D overlay compiler.

Called By:	cig_config	
Routines Called:	make_m1_overlays make_m2_overlays printf	
Parameters:	MSG_OVERLAY_SETUP VIEWPORT_PARAMETERS	*pmsg *pview
Returns:	none	

2.2.1.10 process_vflags.c

The process_vflags function processes system view flags and branch values for conditional nodes. This function is called when the message from the Simulation Host is

MSG_VIEW_FLAGS. It is called by cig_config to put the initial view flags in the configuration tree, and by simulation to update the view flags during runtime.

System view flags are used to turn CRT monitors on and off, and to control viewing modes such as thermal/daylight TV. The branch values indexed by the *branch_index* for all conditional nodes in the configuration tree are also stored in the system view flags array.

The function call is **process_vflags(imsg, top_of_configtree, db)**, where:

imsg is a pointer to the MSG_VIEW_FLAGS message
top_of_configtree is a pointer to the root configuration node
db is the double-buffer memory current base pointer

process_vflags the following:

- Sets up the view modes for DTP.
- If a Force board is present, puts the name of the new color lookup table in Force memory. (The table is downloaded to GSP memory by the forcetask.)
- Processes the view flags and branch values.
- Loads the view flags into the T&C (Timing and Control) board.
- If a Force board is present, puts the video control commands in Force memory. (These commands are downloaded to GSP memory by the forcetask.)

Called By:	cig_config read_configfile simulation	
Routines Called:	none	
Parameters:	CONFIGURATION_NODE I4P INT_4	*top_of_configtree imsg db
Returns:	none	

2.2.1.11 process_vppos.c

The process_vppos function sets up the simulated vehicle's position (the x, y, and z coordinates of its centroid) in the world. This position is used by rowcol_rd to determine whether new load modules need to be read into active area memory. It is also used by local_terrain when preparing local terrain messages for the Simulation Host.

This function is called by confignode_setup when creating a world/hull matrix node (a child of the root node). It is also called by simulation whenever a word/hull matrix node is updated (e.g., in response to a matrix message).

The function call is **process_vppos(config_node, matrix, db)**, where:

config_node is a pointer to the configuration node (always a world/hull node)

matrix is the node's new matrix
db is the double-buffer memory current base pointer

The simulated vehicle's position is stored in an array. This structure allows for multiple vehicles. At the current time, only one simulation vehicle is supported; therefore, there is only one element in the array. The viewport positions array is pointed to by the root node's sibling pointer.

process_vppos takes the matrix provided by the Simulation Host and converts it into world coordinates. The algorithm used to do this depends on the matrix type, as follows:

RTS4x3_TYPE

Given a world-to-view matrix of:

```
| r00 r01 r02 0 |
| r10 r11 r12 0 |
| r20 r21 r22 0 |
| tx  ty  tz  1 |
```

The location of the vehicle in the world is:

vppos.x = -(tx,ty,tz)*(r00,r01,r02)
vppos.y = -(tx,ty,tz)*(r10,r11,r12)
vppos.z = -(tx,ty,tz)*(r20,r21,r22)

RTS3x3_TYPE

The location of the vehicle in the world is:

vppos.x = viewpos->x = matrix.rts3x3.translation.x
vppos.y = viewpos->y = matrix.rts3x3.translation.y
vppos.z = viewpos->z = matrix.rts3x3.translation.z

ROT2x1_TYPE

No conversion is required.

Called By:	confignode_setup simulation	
Routines Called:	none	
Parameters:	CONFIGURATION_NODE MTXUNION INT_4	*config_node matrix db
Returns:	none	

2.2.1.12 read_configfile.c

The functions in read_configfile.c repackage configuration file data into SIM-to-CIG messages. This allows a configuration tree to be built from commands in an ASCII file instead of messages from a Simulation Host. The ASCII file is created off-line and loaded onto the CIG.

The functions in read_configfile.c are:

- read_configfile
- WORD_fscanf
- REAL4_fscanf
- STRING_fscanf
- parser

read_configfile is the driving function. The other functions are used by read_configfile to interpret the data in the configuration file.

> *Note:* *The MSG_GEN_CONFIGTREE message, which would cause read_configfile to be invoked, is not currently implemented. Therefore, none of the functions in read_configfile.c are currently used.*
>
> *An ASCII configuration file can be read by flea_init_cig_sw (in the Flea CSC) for stand-alone use.*

2.2.1.12.1 read_configfile

The read_configfile function reads data from the configuration file and transforms it into SIM-to-CIG messages.

The function call is **read_configfile(filename)**, where *filename* is the name of the configuration file.

read_configfile does the following:

- Opens the specified file.
- Builds the Simulation Host-type message packet.
- Processes each node message; calls confignode_setup to create each node entry.
- Processes each viewport parameter message; calls viewport_setup to create each viewport entry.
- Processes the view flags message; calls process_vflags to create the view flags and the branch value array.
- Closes the file.
- Calls fill_tree to add the graphics path parameters to the tree.

The function returns 1 (SUCCEED) if the file is read and translated successfully. It returns NULL if the specified file could not be opened.

Called By:	none
Routines Called:	close confignode_setup fill_tree getch parser process_vflags

REAL4_fscanf
STRING_fscanf
viewport_setup
WORD_fscanf
XOPEN

Parameters:	char	filename
Returns:	err_code	

2.2.1.12.2 WORD_fscanf

The WORD_fscanf routine searches a file character-by-character looking for a digit. When it finds a digit, it returns the number (WORD type) to which the digit belongs.

The function call is **WORD_fscanf(hex_flag, fp)**, where:

hex_flag identifies the type of digit (DECIMAL or HEX)
fp is a unique identifier associated with the file to be read

Called By:	read_configfile	
Routines Called:	getch isdigit isspace string_to_int	
Parameters:	INT_4 BOOLEAN	fp hex_flag
Returns:	word	

2.2.1.12.3 string_to_int

The string_to_int routine converts a character string to an integer, then returns the result.

The function call is **string_to_int(hex_flag, string)**, where:

hex_flag identifies the type of result desired (DECIMAL or HEX)
string is the string to be converted

Called By:	WORD_fscanf
Routines Called:	isdigit

update_fov does the following:

- Calculates the field-of-view/graphics path and the level-of-detail multiplier.
- Determines which double buffer is being updated this frame.
- Loads the level-of-detail multiplier.
- Initializes values required for the viewspace matrices.
- Calculates each graphics path's *sin_i* and *cos_i*. (These values are required for viewspace matrix calculations.)
- Calls viewspace_mtx to set up the perspective and non-perspective matrices.
- Loads the field-of-view vectors.

Called By:	simulation viewport_setup	
Routines Called:	cos sin tan viewspace_mtx	
Parameters:	CONFIGURATION_NODE REAL_4 REAL_4 REAL_4 INT_4	*config_node SIM_lod fov_i fov_j db
Returns:	none	

2.2.1.13.2 viewspace_mtx

The viewspace_mtx function generates perspective view matrices for use by the Polygon Processor, and non-perspective view matrices for use by DTP.

The function call is **viewspace_mtx(cos_i, sin_i, itan_i, itan_j, perspect_mtx, nperspect_mtx)**, where:

cos_i is the cosine of the graphics path
sin_i is the sine of the graphics path
itan_i is the inverse of the tangent of the fov angle i (horizontal)
itan_j is the inverse of the tangent of the fov angle j (vertical)
perspect_mtx is a pointer to the perspective view matrix
nperspect_mtx is a pointer to the non-perspective view matrix

If load module blocking is enabled, viewspace_mtx scales perspective matrices for the larger active area memory.

Called By: update_fov

Routines Called:	getmatrix id_4x3mtx make_p_nt r4mat_dump (in debug mode only) rotate_z_nt swap_axis	
Parameters:	REAL_4 REAL_4 REAL_4 REAL_4 MAT_UNIT MAT_UNIT	cos_i sin_i itan_i itan_j *perspect_mtx *nperspect_mtx
Returns:	none	

2.2.1.14 update_rez.c

The update_rez function updates the screen resolution in the graphics path parameter structures if a new value is provided by the Simulation Host during runtime.

The function call is **update_rez(config_node, db)**, where:

config_node is a pointer to the configuration node
db is the double-buffer memory current base pointer

Called By:	viewport_setup	
Routines Called:	none	
Parameters:	CONFIGURATION_NODE INT_4	*config_node db
Returns:	none	

2.2.1.15 vec_dump.c

The functions in the vec_dump.c CSU can be used to dump vectors to the standard output (stdout). These functions are:

- r4vec_dump
- r8vec_dump

2.2.1.15.1 r4vec_dump

The r4vec_dump function dumps a vector to stdout.

The function call is **r4vec_dump(str, v)**, where:

str is a string to output to identify the vector (currently undefined)
v is the vector

This function is not currently used.

Called By:	none	
Routines Called:	printf	
Parameters:	char	*str
	REAL_4	v[3]
Returns:	none	

2.2.1.15.2 r8vec_dump

The r8vec_dump function dumps a vector to stdout.

The function call is **r8vec_dump(str, v)**, where:

str is a string to output to identify the vector (currently undefined)
v is the vector

This function is not currently used.

Called By:	none	
Routines Called:	printf	
Parameters:	char	*str
	REAL_8	v[3]
Returns:	none	

2.2.1.16 viewport_setup.c

The functions in the viewport_setup.c CSU are used to create viewport parameter entries in the configuration tree. These functions are:

- viewport_setup
- calc_paths
- viewport_init

2.2.1.16.1 viewport_setup

The viewport_setup function creates and initializes the viewport parameter entries for the terminal nodes in the configuration tree. viewport_setup is called by cig_config when the message from the Simulation Host is MSG_VIEWPORT_STATE.

The function call is **viewport_setup(imsg, top_of_configtree, top_of_view_entries, top_of_path_entries, db)**, where:

imsg is a pointer to the MSG_VIEWPORT_STATE message
top_of_configtree is a pointer to the configuration tree
top_of_view_entries is a pointer to the viewport parameters
top_of_path_entries is a pointer to the graphics path parameters
db is the double-buffer memory current base pointer

viewport_setup does the following:

- Sets a pointer to the owner configuration node.
- Unpacks the message packet from the Simulation Host.
- Sets up a pointer to the viewport positions array.
- Calls calc_paths to determine how many graphics paths are needed, based on the viewport resolution.
- Sets up a local graphics path counter.
- Updates the path count if processing a new viewport.
- Makes sure enough graphics paths are available.
- Calculates the horizontal and vertical field-of-view angles for each graphics path.
- Calculates the screen resolution for each graphics path.
- Loads AAM addresses for the level-of-detail multiplier, viewing range (farthest distance that can be seen), and near plane (closest distance that can be seen).
- Fills in the viewport entry pointer, sibling pointer, path id, and AAM address to field-of-view vectors in the graphics path entries.
- Calls update_fov to fill in the fields related to field of view.
- Updates the viewport and graphics path entry indices.

The function returns 1 if the viewport parameters are added to the configuration tree successfully. It returns NULL if there are not enough graphics paths available.

Called By: cig_config
read_configfile

	isxdigit	
Parameters:	char BOOLEAN	string[] hex_flag
Returns:	result	

2.2.1.12.4 REAL4_fscanf

The REAL4_fscanf routine searches a file character-by-character looking for a digit. When it finds a digit, it returns the number (REAL_4 type) to which the digit belongs.

The function call is **REAL4_fscanf(fp)**, where *fp* is a unique identifier associated with the file to be read.

Called By:	read_configfile	
Routines Called:	atof getch isdigit	
Parameters:	INT_4	fp
Returns:	real4	

2.2.1.12.5 STRING_fscanf

The STRING_fscanf routine searches a file character-by-character looking for a lower- or uppercase alphabetic character. When it finds a legal character, it returns the string to which the character belongs.

The function call is **STRING_fscanf(fp, string)**, where:

fp is a unique identifier associated with the file to be read
string is a pointer to the returned string

Called By:	read_configfile	
Routines Called:	getch isalpha	
Parameters:	INT_4 char	fp string[]

Returns: none

2.2.1.12.6 parser

The parser function parses a configuration file for the configuration messages used by read_configfile to build the corresponding configuration tree. The function returns the next message from the configuration file. Usually, it determines the message from reading just the first character; it reads additional characters if necessary.

The function call is **parser(fp)**, where *fp* is a unique identifier associated with the configuration file.

Called By:	read_configfile	
Routines Called:	STRING_fscanf	
Parameters:	INT_4	fp
Returns:	cmd_line	

2.2.1.13 update_fov.c

The functions in update_fov.c fill in the field-of-view (fov) fields in the graphics path parameters and the viewport parameter entries. They also generate perspective and non-perspective view matrices. These functions are:

- update_fov
- viewspace_mtx

2.2.1.13.1 update_fov

The update_fov function fills in the fov-related fields in the graphics path parameters and the viewport parameter entries. This function is called by viewport_setup during viewport configuration. It is also called by simulation to change field-of-view parameters during runtime.

The function call is **update_fov(config_node, fov_i, fov_j, SIM_lod, db)**, where:

config_node is a pointer to the configuration node
fov_i is the horizontal field-of-view angle
fov_j is the vertical field-of-view angle
SIM_lod is the level-of-detail multiplier to be applied to all non-terrain objects
db is the double-buffer memory current base pointer

Routines Called:	aam_malloc calc_paths mtxcpy update_fov update_rez	
Parameters:	WORD CONFIGURATION_NODE VIEWPORT_PARAMETERS GRAPHICS_PATH_PARAMETERS INT_4	*imsg *top_of_configtree *top_of_view_entries *top_of_path_entries db
Returns:	NULL 1 (SUCCEED)	

2.2.1.16.2 calc_paths

The calc_paths function calculates how many graphics paths are required. For the 120TX, this is based on the desired viewport resolution.

The function call is **calc_paths(resolution_i, resolution_j)**, where:

resolution_i is the number of pixels to display per row (horizontal)
resolution_j is the number of pixels to display per column (vertical)

The function returns the number of graphics paths required.

Called By:	viewport_setup	
Routines Called:	none	
Parameters:	REAL_4 REAL_4	resolution_i resolution_j
Returns:	graphics_paths	

2.2.1.16.3 viewport_init

The viewport_init function resets all static variables used by the viewport_setup function. These variables are the graphics path count, view entry index, path entry index, and maximum graphics paths count. This function is called by init_configtree before viewport_setup is called by cig_config.

The function call is **viewport_init()**.

Called By:	init_configtree
Routines Called:	none
Parameters:	none
Returns:	none

2.2.2 DTP Command Generator

The DTP (Data Traversal Processor) Command Generator translates the viewport configuration tree generated by the real-time software into the commands required to drive the graphics hardware. It generates DTP hardware commands (processor and channel initialization code) from the viewport configuration tree, then downloads these commands to the DTP CPU. The DTP then determines what data is to be sent to the 9U graphics channel.

The DTP is a micro-coded processor board that does the following:

- Looks through active area memory for DTP commands.
- Computes viewpoint positions for vectors.
- Computes world-to-viewpoint matrices for each viewport.
- Performs field-of-view and level-of-detail tests on models and special effects.
- Sends data to the Polygon Processor.

The Polygon (Poly) Processor is a special-purpose floating point processor that does the following:

- Transforms polygons from world coordinates to viewspace coordinates.
- Eliminates back-facing polygons.
- Clips polygons that fall partially outside of the viewing pyramid.
- Fills polygons with colored or textured pixels.
- Perspectively projects polygons onto the screen.

The DTP is controlled through the DTP commands it finds in active area memory. These commands are placed in active area memory by the DTP Command Generator. The DTP reads one buffer in double-buffer memory while the real-time software updates the other. Each frame, the two processes switch buffers.

The DTP Command Generator uses the Runtime Command Library (RCL) to generate DTP commands. The RCL is a set of software functions that support the configuration of lists of runtime commands for both the DTP and the Poly Processor. The RCL is responsible for working with the complex data structures in the DTP — the DTP Command Generator simply specifies the command and provides the data required for the command. The RCL also maintains addressing and data sizing information.

The interface between the DTP Command Generator functions and the RCL is implemented via command-specific macros. Each DTP command is supported by one or more macros. These macros are named in the form *dtp_xyz*, where *xyz* identifies the DTP command or a version of a command. Similarly, macros that support Poly Processor commands are named in the form *poly_xyz*. The DTP Command Generator function calls the appropriate macro and passes it the data required for the selected command. The macro in turn calls the appropriate RCL routine and passes it the command parameters. The RCL routine then generates the actual DTP command and places it in active area memory.

The DTP-RCL macros are defined in the rcinclude.h file. Refer to Appendix B for a list of these macros.

Figure 2-5 identifies the CSUs in the DTP Command Generator component of the UPSTART CSC. These CSUs are described in this section.

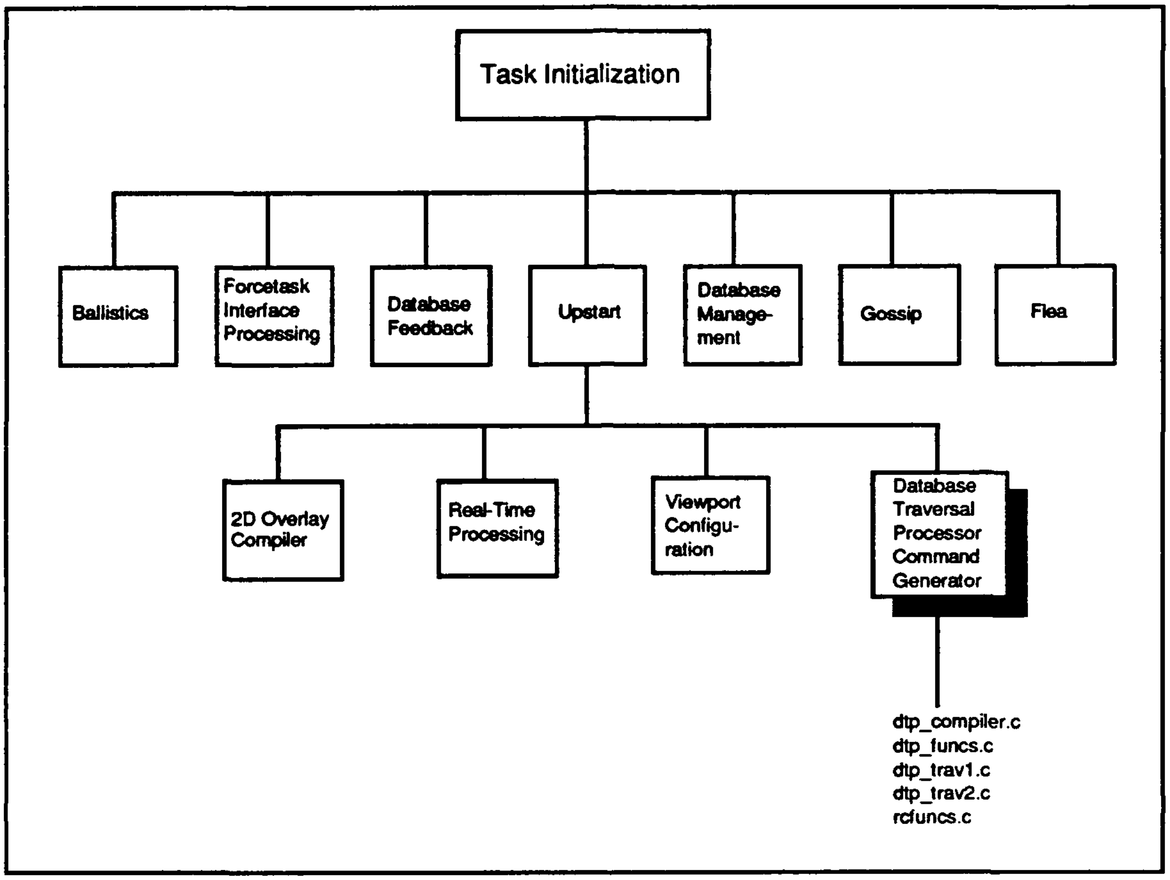


Figure 2-5. DTP Command Generator CSUs

2.2.2.1 dtp_compiler.c

The dtp_compiler function is the driving function for generating DTP hardware commands from the viewport configuration tree.

The function call is **dtp_compiler(root, offset)**, where:

root is a pointer to the configuration node
offset is the number of bytes of DTP code

dtp_compiler does the following:

- Initializes the runtime command library (RCL).
- Allocates data pointers for model processing.
- Initializes the DTP stack.
- Calls dtp_trav1 to traverse the configuration tree for processor initialization.
- Runs the RCL patch utility to patch any missing addresses and word counts.
- Reinitializes the RCL and DTP stacks.
- Calls dtp_trav2 to traverse the configuration tree for channel initialization.
- Runs the RCL patch utility again.

- Prints DTP memory use data.

The function returns 1 if the commands are generated successfully, or 0 if either dtp_trav1 or dtp_trav2 fails.

Called By:	cig_config	
Routines Called:	dtp_trav1 dtp_trav2 init_dtp_stacks printf rcl_init_adrs rcl_init_stack rcl_patch_adrs rcl_rtn_adrs rcl_set_errptr	
Parameters:	CFG_NODE WORD	*root offset
Returns:	0 (FAIL) 1 (SUCCEED)	

2.2.2.2 dtp_funcs.c

The functions in the dtp_funcs.c CSU are called by dtp_trav1 to (1) manage the node stack it uses to traverse the configuration tree, and (2) allocate DTP user memory. These functions are:

- push_node
- pop_node
- what_node_on_stack
- init_dtp_stacks
- dtp_malloc
- dtp_malloc_init

2.2.2.2.1 push_node

The push_node function takes a configuration node pointer as input and places it on the stack. It also checks for and reports node stack overflows.

The function call is **push_node(node_ptr)**, where *node_ptr* is a pointer to the configuration node to be pushed on the top of the stack.

Called By: dtp_trav1

Routines Called:	printf	
Parameters:	CONFIGURATION_NODE	*node_ptr
Returns:	none	

2.2.2.2.2 pop_node

The pop_node function returns the configuration node pointer from the top of the stack. If the node stack is empty, pop_node returns 0; this tells dtp_trav1 that the stack has been processed completely.

The function call is **pop_node()**.

Called By:	dtp_trav1
Routines Called:	none
Parameters:	none
Returns:	node stack pointer 0

2.2.2.2.3 what_node_on_stack

The what_node_on_stack function returns the node index of the node on top of the stack.

The function call is **what_node_on_stack(empty)**, where *empty* is the value to be returned if the stack is empty.

Called By:	dtp_trav1	
Routines Called:	none	
Parameters:	WORD	empty
Returns:	node_index empty	

2.2.2.2.4 init_dtp_stacks

The init_dtp_stacks function initializes the DTP stack pointers to the top of the stack.

The function call is **init_dtp_stacks()**.

Called By:	dtp_compiler	
Routines Called:	none	
Parameters:	none	
Returns:	none	

2.2.2.2.5 dtp_malloc

The dtp_malloc function allocates DTP memory. This function is called by dtp_trav1 to allocate memory for configuration node matrices.

The function call is **dtp_malloc(count)**, where *count* is the amount of memory to be allocated.

The function returns 0 if successful. It returns the current DTP user pointer (as *give_away*) if insufficient memory is available.

Called By:	dtp_trav1	
Routines Called:	none	
Parameters:	INT_2	count
Returns:	0 give_away	

2.2.2.2.6 dtp_malloc_init

The dtp_malloc_init function initializes the portion of DTP allocated as user space. It sets the DTP user pointer to the first available memory location, which is defined in ecompiler1.h. dtp_trav1 calls this function before it starts traversing the configuration tree.

The function call is **dtp_malloc_init()**.

Called By: dtp_trav1

Routines Called: none

Parameters: none

Returns: none

2.2.2.3 dtp_trav1.c

The dtp_trav1 function function traverses the configuration tree to generate processor initialization codes. It traverses each node in the configuration tree by placing the root node on the stack and then processing the stack until it is empty. When a node is popped from the stack, any matrix concatenation commands or bit tests are performed for that node, based on the node's type. The node's children and siblings are then placed on the stack such that the order of processing is the node, the node's children, and the node's siblings.

dtp_trav1 uses the routines in dtp_funcs.c to access and manage the node stack. It uses the dtp_* macros (defined in Appendix B) to communicate with the RCL to generate the actual commands for the hardware.

The function call is **dtp_trav1(node)**, where *node* is a pointer to the root configuration node. dtp_trav1 does the following:

- Calls dtp_malloc_init to initialize the DTP user space.
- Uses various dtp_* macros to load the following:
  - Channel status and channel pointers at DTP location 0.
  - List of final processing.
  - Flush and dynamic pointer tables.
  - Calibration branch mask.
  - Cloud bottom and top branch masks (if enabled).
  - Daylight TV thermal word.
  - View mode word for each channel.
  - System view flags and branch values.
  - Current time set in simulation.
- Processes each node in the tree to generate the matrix concatenations and bit tests for each path, as follows:
  - Calls push_node to push the root child 0 on the stack.
  - Calls pop_node to pop each node from the stack in turn.
  - Calls rcl_set_label to set a label for the node.
  - Validates the node's parent pointer.
  - For a matrix node:
    - Allocates DTP memory for the node's matrix.
    - Concatenates the matrix with the parent's matrix.
  - For a branch/matrix node:
    - Test the node's branch value.
    - Allocate DTP memory for the node's matrix.
    - If the branch value is true, load the node's matrix or concatenate it with the parent's matrix.

    * If the branch value is false, load the parent's matrix.
  - For a branch (conditional) node:
    * Test the node's branch value.
    * Load the parent's matrix.
  - Push the node's siblings and children onto the stack.
- Performs initial data traversal.
- Prepares system post-processing pointers and displays the post-processing addresses for static vehicles, dynamic vehicles, and effects.
- Allocates space for the current time to support time-base commands.
- Calls rcl_data to generate a command to indicate a separation of initialization and channel processing.

The function returns 1 if successful. It returns 0 if it detects an illegal parent pointer or an invalid node type.

Called By: dtp_compiler

Routines Called:
dtp_bnz
dtp_bru
dtp_brus
dtp_end
dtp_lwd
dtp_lwds
dtp_malloc
dtp_malloc_init
dtp_mmpst
dtp_mwd
poly_flu
pop_node
printf
push_node
rcl_data
rcl_rtn_adrs
rcl_set_errptr
rcl_set_label
what_node_on_stack

Parameters: CONFIGURATION_NODE *node

Returns:
0
1

2.2.2.4 dtp_trav2.c

The dtp_trav2 function traverses the configuration tree to generate channel initialization codes.

The function call is **dtp_trav2(node)**, where *node* is a pointer to the root configuration node. dtp_trav2 does the following:

- Saves the beginning path location.
- For a branch (conditional) node:
  - Tests the condition.
  - Traverses the true path.
- For a matrix node that is the terminal node in a traversal path (i.e., a node that has viewport parameters):
  - Calculates the channel base address.
  - Loads the channel parameters, field-of-view vectors, viewpoint position, level-of-detail multiplier, and far plane.
  - Multiplies the hull-to-view matrix for DTP use.
  - Calculates bounding plane normals.
  - Calculates the base load module.
  - Outputs the channel toggle command if the channel is secondary.
  - Outputs the perspective matrix.
  - Outputs the screen constants.
  - Tests for calibration output for all screens.
  - Outputs the gun overlay if bit 0 of the node's branch mask is set.
- For the root node:
  - Saves the matrix and forms the stamp word.
  - Calls the cloud top and bottom models, if enabled.
- Pre-processes models:
  - Creates *output_direct* for the node's matrix.
  - Outputs the gun barrel overlay if bit 1 of the node's branch mask is set.
  - For a branch node, sets the branch mask.
- Prepares the system pre-processing pointers and displays the pre-processing addresses for dynamic vehicles, static vehicles, and effects.
- Saves common poly command data.

The function always returns 1.

Called By:	dtp_compiler
Routines Called:	dtp_blm dtp_bnz dtp_bpc dtp_bru dtp_brus dtp_brz dtp_end dtp_lwds dtp_mmpst dtp_osd dtp_owd dtp_owds dtp_subs poly_fsw poly_rm1 poly_sm1 poly_tog printf rcl_rtn_adrs

rcl_set_errptr
rcl_set_label
rcl_stuff_data

Parameters: CONFIGURATION_NODE *node

Returns: 1

2.2.2.5 rcfuncs.c

The functions in the rcfuncs.c CSU are used to work with the Runtime Command Library (RCL). These functions are:

- rcl_init_stack
- rcl_push
- rcl_pop
- rcl_patch_adrs
- rcl_set_errptr
- rcl_init_adrs
- rcl_rtn_adrs
- rcl_set_label
- rcl_set_cntlbl
- rcl_lblcmd
- rcl_command
- rcl_component
- rcl_data
- rcl_stuff_data

This CSU also defines the following macros used by the RCL functions. These macros are described in Appendix B.

- ERRMSG
- ROOM4LABEL
- ROOMCHECK
- INCR_COMPONENT

The RCL labeling utility removes the need for the programmer to maintain addressing and data size information as a command sequence is constructed. The programmer can write runtime code and label only data that is unknown. All labels (defined as single-integer values) must uniquely identify one location in the code. As the library generates the runtime commands, it places any unknown information onto a patch stack. When the library encounters a label, it stores the location of the label for use in patching the stack. The rcl_patch_adrs function scans the list of unknown data and patches the missing addresses and word counts.

Use of the patching utility requires a stack area for maintaining the unresolved addresses, counts, and labels. The rcl_init_stack function is used to initialize the stack.

Most labels are used to identify a location in active area memory. Some labels are branch labels where DTP branch commands change the direction in which the DTP is processing messages. DTP output commands reference a location where the data begins. For these commands, the calling function specifies a unique label to identify the branch of output data, and uses the rcl_set_label function to identify the location. These locations are patched with the supplied addresses when the rcl_patch_adrs function is executed.

Set count labels are labels that are used to identify the size of a data segment rather than the location of command data.

The DTP has several output commands that require a word count value in order for the DTP to pass the correct amount of data to the Poly Processor. Usually, there are two ways to accomplish this:

- If the exact amount of data that can be sent is known, the DTP output command using the function that has data start label and word count parameters can be used.
- If the data size is not known, the command can be implemented using the set count function. Rather than specifying a word count for the command, a set count label is defined. When generating the data, rcl_set_label is executed to identify the beginning of the data. After generating the data, rcl_set_cntlbl is executed to specify the start and end labels, and the set count label is loaded with the word count of the data segment. When rcl_patch_adrs is executed, the output count is patched with the data segment size.

The DTP supports two addressing modes: absolute and relative. In absolute mode, the address is the actual AAM address for the branch or data. In relative mode, the address is an offset that is added to the current location to locate the branch or data.

2.2.2.5.1 rcl_init_stack

The rcl_init_stack function initializes the unresolved address, count, and label stack.

The function call is **rcl_init_stack(min_stack, max_stack)**, where:

min_stack is the minimum stack address
max_stack is the maximum stack address

Called By:	dtp_compiler	
Routines Called:	none	
Parameters:	WORD WORD	*min_stack *max_stack
Returns:	none	

2.2.2.5.2 rcl_push

The rcl_push function adds a patch location to the patch stack.

The function call is **rcl_push(adr, lbladr, name)**, where:

adr is the physical memory address the patch is to be made in
lbladr is the physical memory address the label for the patch is in
name is the name of the calling routine

The function returns 0 if successful, or 1 if the stack is full.

Called By:	rcl_lblcmd	
Routines Called:	ERRMSG	
Parameters:	WORD WORD char	*adr *lbladr *name
Returns:	0 1	

2.2.2.5.3 rcl_pop

The rcl_pop function removes a patch location from the patch stack.

The function call is **rcl_pop(adr, lbladr, name)**, where:

adr is the physical memory address the patch is to be made in
lbladr is the physical memory address the label for the patch is in
name is the name of the calling routine

The function returns 0 if successful, or 1 if the stack is empty.

Called By:	rcl_patch_adrs	
Routines Called:	ERRMSG printf	
Parameters:	WORD WORD char	*adr *lbladr *name
Returns:	0 1	

2.2.2.5.4 rcl_patch_adrs

The rcl_patch_adrs function removes remaining entries from the patch stack one at a time. It patches the stored location with the associated label location and processes the stack until it is empty. This function patches both absolute and relative addresses.

The function call is **rcl_patch_adrs()**.

Called By:	dtp_compiler	
Routines Called:	ERRMSG printf rcl_pop	
Parameters:	none	
Returns:	none	

2.2.2.5.5 rcl_set_errptr

The rcl_set_errptr function can be used to specify a character string to be output with every RCL error message. This string can help localize the source of the error.

The function call is **rcl_set_errptr(adr)**, where *adr* is the error string.

Called By:	dtp_compiler dtp_trav1 dtp_trav2	
Routines Called:	none	
Parameters:	char	*adr
Returns:	none	

2.2.2.5.6 rcl_init_adrs

The rcl_init_adrs function initializes values for shared addressing variables.

The function call is **rcl_init_adrs(bld_adr, aam_adr, byte_count)**, where:

bld_adr is the memory location to store the RCL commands

aam_adr is the AAM location corresponding to the *bld_adr*
byte_count is the number of bytes available for RCL commands, starting at *bld_adr*

Called By:	dtp_compiler	
Routines Called:	none	
Parameters:	WORD WORD WORD	*bld_adr aam_adr byte_count
Returns:	none	

2.2.2.5.7 rcl_rtn_adrs

The rcl_rtn_adrs function returns the current values of RCL addressing values, as defined in *init_addressing*.

The function call is **rcl_rtn_adrs(bld_adr, aam_adr, byte_count)**, where:

bld_adr is the address to return the memory location to store the RCL commands
aam_adr is the address to return the AAM location corresponding to the *bld_adr*
byte_count is the address to return the number of bytes available for RCL commands

Called By:	dtp_compiler dtp_trav1 dtp_trav2	
Routines Called:	none	
Parameters:	WORD WORD WORD	**bld_adr *aam_adr *byte_count
Returns:	none	

2.2.2.5.8 rcl_set_label

The rcl_set_label function specified that a given label refers to the current location in active area memory (the location in *rcl_aam_adr*).

The function call is **rcl_set_label(label)**, where *label* is the label to set with the AAM location.

Called By:	dtp_trav1 dtp_trav2	
Routines Called:	ERRMSG printf ROOM4LABEL	
Parameters:	WORD	label
Returns:	none	

2.2.2.5.9 rcl_set_cntlbl

The rcl_set_cntlbl function identifies a section of data for output. The function stores in *cnt_label* the number of words from the address stored in *label* to the current AAM address. Output commands that refer to the set count label *cnt_label* are patched with this data.

The function call is **rcl_set_cntlbl(label, cnt_label)**, where:

label is a previously set label that identifies the beginning of the data
cnt_label is the label associated with an output count

Called By:	none	
Routines Called:	ERRMSG printf ROOM4LABEL	
Parameters:	WORD WORD	label cnt_label
Returns:	none	

2.2.2.5.10 rcl_lblcmd

The rcl_lblcmd function generates a DTP label command.

The function call is **rcl_lblcmd(name, wd_cnt, id, rel, lbl)**, where:

name is a pointer to the name of the calling routine
wd_cnt is the total number of words the function will generate for the command
id is the command id value
rel is the relative addressing flag
lbl is the label the command branch value is associated with

rcl_lblcmd does the following:

- Calls ROOMCHECK to make sure there is room for the command.
- Calls ROOM4LABEL to make sure there is room for the label.
- Pushes the address and label address on the stack to patch.
- Saves the correct addressing.
- Copies the additional data.
- Updates memory data.

When rcl_lblcmd places the command location on the stack, *rel* is stored as the branch data. *rel* is set to 90 for absolute addressing, and is set to *rcl_aam_adr* for relative addressing. When patching occurs, this value is subtracted from the patch label to generate the relative or absolute value.

If *wd_cnt* is greater than 1, the data following *lbl* on the function stack is appended to the command.

Called By:

dtp_bcn
dtp_bcnr
dtp_bcz
dtp_bczr
dtp_bdgr
dtp_bdlr
dtp_bgn
dtp_bgz
dtp_bnz
dtp_bnzr
dtp_bru
dtp_brur
dtp_brz
dtp_brzr
dtp_fov
dtp_fovr
dtp_gdc
dtp_gdci
dtp_gdcir
dtp_gdcn
dtp_gdcnr
dtp_gdcr
dtp_lmi
dtp_lmir
dtp_lod
dtp_lodr
dtp_lwd
dtp_lwdr
dtp_osd
dtp_owd
dtp_owdsc
dtp_owr
dtp_owrsc
dtp_sub
dtp_subr

dtp_tbdr
dtp_tbrr
poly_efs
poly_efsr

Routines Called:	rcl_push ROOM4LABEL ROOMCHECK	
Parameters:	char WORD BYTE WORD WORD	*name wd_cnt id rel lbl
Returns:	none	

2.2.2.5.11 rcl_command

The rcl_command function generates a DTP command with no label.

The function call is **rcl_command(name, wd_cnt, id, data)**, where:

name is a pointer to the routine name
wd_cnt is the total number of command WORDs
id is the command id value
data is the data for the command

rcl_command does the following:

- Calls ROOMCHECK to make sure there is room for the command.
- Copies the data.
- Puts the command id in memory.
- Updates memory data.

Called By:

dtp_bcnrs
dtp_bcns
dtp_bczrs
dtp_bczs
dtp_bdgrs
dtp_bdlrs
dtp_bgns
dtp_bgzs
dtp_blm
dtp_bnzrs
dtp_bnzs
dtp_bpc
dtp_bpcx
dtp_brurs

dtp_brus
dtp_brzrs
dtp_brzs
dtp_dot
dtp_elm
dtp_end
dtp_fovrs
dtp_fovs
dtp_gdcirs
dtp_gdcis
dtp_gdcnrs
dtp_gdcns
dtp_gdcrs
dtp_gdcs
dtp_gr
dtp_lmirs
dtp_lmis
dtp_lodrs
dtp_lods
dtp_lwdrs
dtp_lwds
dtp_mml
dtp_mmpre
dtp_mmpst
dtp_mwd
dtp_ngc
dtp_oio
dtp_oos
dtp_osds
dtp_owds
dtp_owo
dtp_owrs
dtp_rc
dtp_subrs
dtp_subs
dtp_tbc
dtp_tbdrs
dtp_tbrrs
poly_flu
poly_fsw
poly_lmf
poly_lsc
poly_mmf
poly_rm1
poly_rm2
poly_rm3
poly_rm4
poly_sm1
poly_sm2
poly_sm3
poly_sm4
poly_tog

Routines Called:	ROOMCHECK	
Parameters:	char	*name
	WORD	wd_cnt
	BYTE	id
	WORD	data
Returns:	none	

2.2.2.5.12 rcl_component

The rcl_component function generates a Poly Processor component command.

The function call is **rcl_component(name, wd_cnt, incr, id, bal, lt, data)**, where:

name is a pointer to the name of the calling routine
wd_cnt is the total number of words the function will generate for the command
incr is the count increment used to initialize component data
id is the command id value
bal is the Ballistics bit
lt is the local terrain bit
data is the first piece of additional data

rcl_component does the following:

- Calls ROOMCHECK to make sure there is room for the command.
- Saves the component pointers for count updates.
- Sets the component id.
- Sets the Ballistics bit if any polygons in the component need to be checked for Ballistics intersections.
- Sets the local terrain bit if any polygons in the component need to be included in the local terrain message sent to the Simulation Host.
- If *wd_cnt* is greater than 1, zeroes the second word of the component.
- Copies the additional data.
- Calls INCR_COMPONENT to update the component's word count, polygon count, and vertex count in the Poly component.
- Updates memory data.

Called By:	poly_bvc
	poly_gc
	poly_pc
	poly_sc
	poly_sci
	poly_sec
Routines Called:	INCR_COMPONENT
	ROOMCHECK

Parameters:	char	*name
	WORD	wd_cnt
	WORD	incr
	BYTE	id
	BYTE	bal
	BYTE	lt
	WORD	data

Returns: none

2.2.2.5.13 rcl_data

The rcl_data function provides additional data for a poly component command.

The function call is **rcl_data(name, wd_cnt, incr, data)**, where:

name is the name of the calling routine
wd_cnt is the total number of words the function will generate for the command
incr is the count increment used to initialize component data
data is the first piece of additional data

rcl_data does the following:

- Calls ROOMCHECK to make sure there is room for the command.
- Copies the data.
- Updates memory data.
- Calls INCR_COMPONENT to update the component's word count, polygon count, and vertex count in the Poly component.

Called By:
poly_ab
poly_inf
poly_poly
poly_sci
poly_stamp
poly_vtxe
poly_vtxl

Routines Called:
INCR_COMPONENT
ROOMCHECK

Parameters:	char	*name
	WORD	wd_cnt
	WORD	incr
	WORD	data

Returns: none

2.2.2.5.14 rcl_stuff_data

The rcl_stuff_data function places a specified number of data words found in a specified location of user memory into successive memory locations. This function is used to add data to the processing path when no function is available to produce the desired effect.

The function call is **rcl_stuff_data(cpf, wd_cnt)**, where:

cpf is a pointer to the data
wd_cnt is the amount of data to copy

rcl_stuff_data does the following:

- Calls ROOMCHECK to make sure there is room for the data.
- Copies the data.
- Updates memory data.

Called By:	dtp_trav2 poly_lmf poly_mmf	
Routines Called:	ROOMCHECK	
Parameters:	WORD WORD	*cpf wd_cnt
Returns:	none	

2.2.3 Real-Time Processing

Real-Time Processing, a major functional component of the UPSTART CSC, is responsible for setting up and running the simulation using messages sent from the Simulation Host.

Upon start-up, the upstart function initializes active area memory, initializes system tasks, verifies that the DR11 communications interface is functional, and loads and starts Ballistics. It then processes messages sent by the Simulation Host to place the CIG in a specified state. The CIG states that can be set are:

Database setup
This state prepares the CIG to run a simulation. If this state is requested, upstart passes control to db_mcc_setup.

File control
This state is used to transfer files to/from the CIG). If this state is requested, upstart passes control to file_control.

Test mode
This state is used to run hardware tests. If this state is requested, upstart passes control to hw_test.

MCC setup
This state prepares the CIG to act as an MCC station. This mode is not currently used.

If database setup is specified, db_mcc_setup processes messages from the Simulation Host to configure the viewports and the 2-D overlays (by initiating Viewport Configuration and the 2-D Overlay Compiler, respectively). db_mcc_setup also loads the terrain database and the dynamic elements database (DED) into active area memory, and processes requests to download trajectory table data. Upon another state change request from the Simulation Host, db_mcc_setup calls simulation to start the simulation.

simulation processes all runtime messages during the simulation. Upon request from the Simulation Host, simulation moves or rotates dynamic vehicles, changes the gun overlays, passes process round and round fired messages to Ballistics, shows effects, adds and removes static vehicles, changes a viewport's field of view or level of detail, changes the view mode, and updates the system view flags. simulation also processes the hit and miss messages returned by Ballistics.

When the Simulation Host sends a message ending the simulation, simulation cleans up and passes control back to db_mcc_setup. db_mcc_setup then initializes the configuration tree and returns control to upstart.

The CSUs in the Real-Time Processing component are identified in Figure 2-6 and are described in this section.

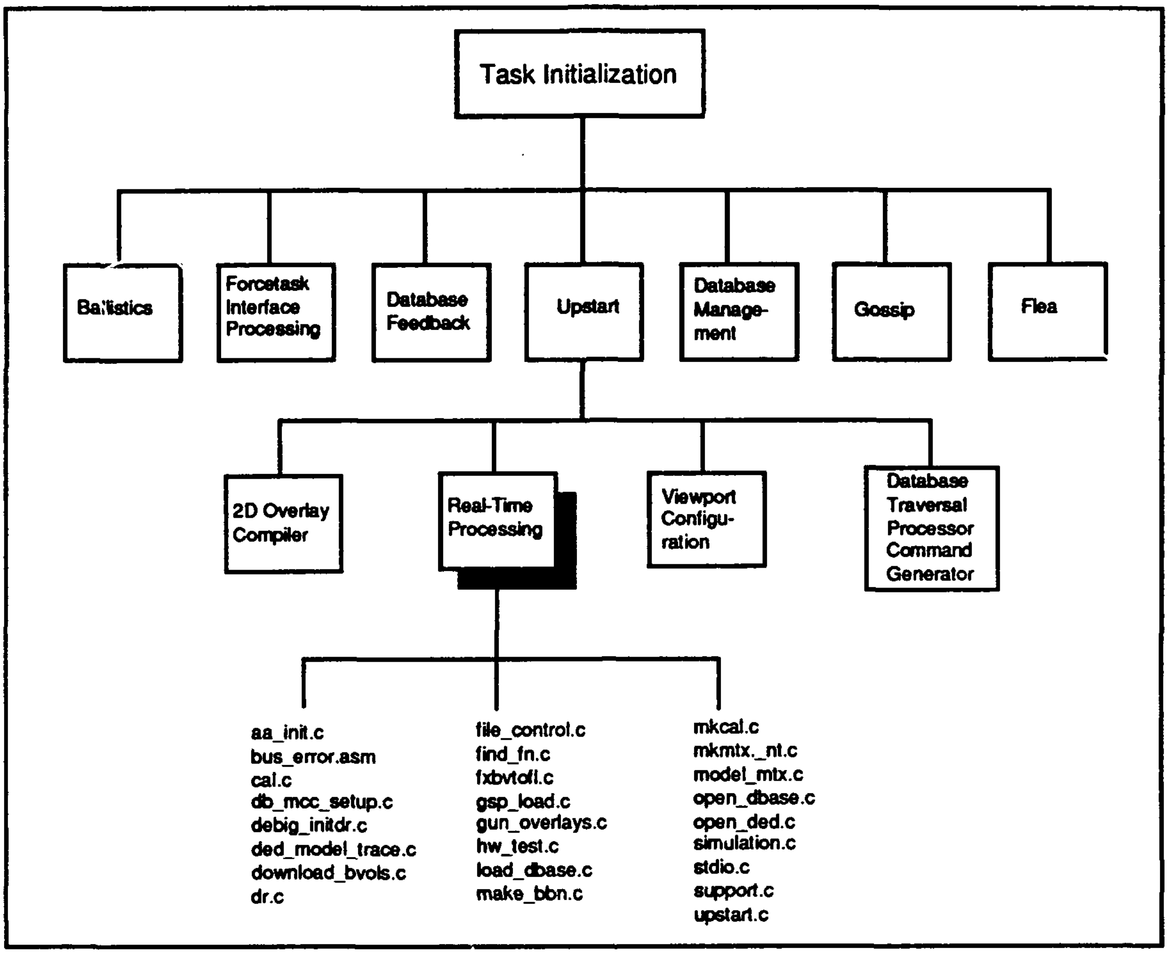


Figure 2-6. Real-Time Processing CSUs

2.2.3.1 aa_init.c (active_area_init)

The aa_init.c CSU contains one function, active_area_init. This function initializes the active area of memory state tables and their related variables. This function is called by upstart on start-up, and by simulation when it receives a CIG Control-Stop message from the Simulation Host.

The function call is **active_area_init()**. active_area_init does the following:

- Clears the system area of active area memory.
- Initializes tanks and other vehicles in the dynamic state table.
- Initializes static tanks and other static vehicles in the static state table.
- Initializes the multiple-frame effects linked list. (This structure is used when showing effects over multiple frames.)

Called By: simulation
upstart

Routines Called: INIT_MTX

Parameters: none

Returns: none

2.2.3.2 bal_routines.c

The functions in the bal_routines.c CSU are not used in the 120TX/T implementation. Information provided on these functions in earlier versions of this document should be disregarded.

2.2.3.3 bus_error.asm

The bus_error function touches a memory location to see if it exists. It is usually used to determine which type of Ballistics board is in the CIG, or to find out whether the CIG contains a Force board.

The function call is **bus_error(address, accesstype)**, where:

address is the test address
accesstype is **b** (byte access), **w** (word access), or **l** (long word access)

The function returns 0 if the memory location exists, or 1 if it does not.

Called By:	apinit load_dbase upstart	
Routines Called:	none	
Parameters:	INT char	address accesstype
Returns:	0 1	

2.2.3.4 cal.c

The cal (calibration menu) function exercises the video monitors by placing a known pattern on all channels. cal presents a menu that lets the Gossip user turn the calibration image or gunner pixel on or off. The user is then able to verify the accuracy of the image and take appropriate measures.

The function call is **cal()**.

Called By:	gossip
Routines Called:	printf unbf_getchar
Parameters:	none
Returns:	none

2.2.3.5 db_mcc_setup.c

The db_mcc_setup function processes messages from the Simulation Host to prepare the CIG system to run a simulation. It can also prepare the CIG to act as an MCC station, although this mode is not currently used. db_mcc_setup is called by upstart when the CIG Control message from the Simulation Host specifies C_DB_SETUP or C_MCC_SETUP.

The function call is **db_mcc_setup(state)**, where *state* is the state the system is to be set up in: C_DB_SETUP (simulation mode) or C_MCC_SETUP (MCC station mode).

db_mcc_setup does the following:

- Initializes trajectory table static variables.
- Processes each message received from the Simulation Host (see table below).
- Returns to upstart when it returns from cig_config or a simulation, or when it detects a CIG-Control Stop message.

The following table summarizes the processing performed by db_mcc_setup in response to each valid message type it receives from the Simulation Host. The first column lists the messages in alphabetical order. The second column briefly describes the purpose of the message (in italics), then lists the major steps performed by db_mcc_setup to process the message.

Message from SIM Host	Processing by db_mcc_setup
MSG_CIG_CTL	*Causes a transition to another performance state.*
C_CIG_CONFIG	Calls gsp_load if there is a Force board and GSP is not initialized; calls cig_config; calls load_dbase.
C_MCC_SIMUL	Calls simulation with state set to C_MCC_SIMUL.
C_NULL	No action.
C_SIMULATION	Calls simulation with state set to C_SIMULATION.
C_START_2D_SETUP	Calls gsp_load if there is a Force board and GSP is not initialized; calls cig_2d_setup if there is a Force board.
C_STOP	Returns to upstart.
MSG_DR11_PKT_SIZE	*Specifies exchange packet parameters.* Sets CIG and SIM exchange packet size, local terrain chunk size, and local terrain message interval.
MSG_END	*Signals end of packet buffer.* Posts a BALLISTICS_MB message if the CIG contains a master Ballistics board; calls EXCHANGE_DATA to send output and receive input buffers.
MSG_FILE_DESCR	*Specifies database to use for simulation.*
DB_SETUP	Calls gsp_load if there is a Force board and GSP is not initialized; calls open_dbase.
DB_DED_SETUP	Sets ded_db_name in global memory.
MSG_TRAJ_ENTRY_XFER	*Downloads an entry in a Ballistics trajectory table.* Sets trajectory table entry data; calls mx_push to push MSG_B0_ADD_TRAJ_ENTRY message onto Ballistics message queue.
MSG_TRAJ_TABLE_XFER	*Sets up a Ballistics trajectory table to be downloaded.* Sets data for trajectory table; calls mx_push to push MSG_B0_TRAJ_TABLE message onto Ballistics message queue.

Called By: upstart

Routines Called:
cig_2d_setup
cig_config
EXCHANGE_DATA
free_configtree
gsp_load
load_dbase
mx_push
open_dbase
printf
sc_post
simulation
SYSERR

Parameters: INT_2 state

Returns: none

2.2.3.6 debug_initdr.c

The debug_initdr function calls the display_packet function (in Gossip) to display the contents of each message in a DR11 exchange packet.

The function call is **debug_initdr()**.

Called By:	EXCHANGE_DATA
Routines Called:	display_packet
Parameters:	none
Returns:	none

2.2.3.7 ded_model_trace.c

The ded_model_trace function traces the Data Traversal Processor (DTP) commands for each dynamic model and adjusts addresses based on the commands.

The function call is **ded_model_trace(ded_size, ded_start_address, model_start_address, gm_end_address)**, where:

ded_size is the amount of memory available for all dynamic models
ded_start_address is the starting location for loading dynamic models
model_start_address is the starting location for a specific model
gm_end_address is the highest address in generic memory

The function returns 0 if successful. It returns -1 if it the model's address is beyond the end of generic memory or before its start address.

Called By:	open_ded	
Routines Called:	printf	
Parameters:	INT_4	ded_size
	INT_4	ded_start_address
	INT_4	model_start_address
	INT_4	gm_end_address
Returns:	0	
	-1	

2.2.3.8 download_bvols.c

The download_bvols function downloads models and bounding volumes to Ballistics.

The function call is **download_bvols(header_P, fd, dev_P, model_type)**, where:

header_P is a pointer to the database header data
fd is an identifier for the file containing the information to be downloaded
dev_P is a pointer to the Ballistics message queue
model_type is BX_DED_MODEL_DIRECTORY

download_bvols does the following:

- Allocates memory to work in.
- Reads the model directory information from the specified database header.
- Builds a structure with the model directory data and passes it to Ballistics by calling mx_push to push a MSG_B0_MODEL_DIRECTORY message onto the Ballistics message queue.
- Reads each model entry in the specified file.
- For each model entry:
  - Builds a structure with the model's data.
  - Passes the structure to Ballistics by calling mx_push to push a MSG_B0_MODEL_ENTRY message onto the Ballistics message queue.
- Reads and validates the bounding volume count from the database header.
- Reads each bounding volume entry from the specified file.
- For each bvol entry:
  - Builds a structure with the bvol's data.
  - Passes the structure to Ballistics by calling mx_push to push a MSG_B0_BVOL_ENTRY message onto the Ballistics message queue.
- Frees the memory it allocated.

The function returns 0 if successful. It returns -1 if the number of bounding volumes is less than 0.

Called By:	open_ded	
Routines Called:	BCOPY free fxbvtofl_020 malloc mx_push printf XLSEEK XREAD	
Parameters:	DB_HDR_DBASE_DATA INT MX_DEVICE BYTE	*header_P fd *dev_P model_type

Returns: 0
-1

2.2.3.9 dr.c

The functions in the dr.c CSU are used to test the DR11 interface between the CIG and the Simulation Host. These functions are:

- dr
- dr_is_okay

2.2.3.9.1 dr

The dr function is a test routine that calls the dr_is_okay function, then loads a file over the DR11 interface when it appears as if the interface is ready to begin communication.

The function call is **dr()**.

Called By:	none
Routines Called:	printf system
Parameters:	none
Returns:	none

2.2.3.9.2 dr_is_okay

The dr_is_okay function looks at absolute memory addresses to attempt to determine whether the DR11 interface is in a safe and stable condition.

The function call is **dr_is_okay()**. dr_is_okay does the following:

- Waits until the DR11 registers show that both the attention bit and the status B bit are not set. This indicates that the cables are plugged in and the Simulation Host is powered up.
- Waits until both the status A and status C bits are set. This indicates that the Simulation Host is waiting to read data.
- Makes sure that at most one event is posted to the *dr_mbox* queue. Removes any excess messages from the queue.

The function returns 1 if it determines that the DR11 is ready to begin communication.

Called By:	dr OPEN_EXCHANGE
Routines Called:	printf sc_lock sc_qinquiry sc_qpend sc_unlock
Parameters:	none
Returns:	1 (TRUE)

2.2.3.10 file_control.c

The file_control function processes messages from the Simulation Host to handle file transfers to and from the Simulation Host, delete files, and produce a CIG disk directory list for the Simulation Host. file_control is called by upstart whenever the state requested by the Simulation Host is C_FILE_XFER.

The function call is **file_control(state)**, where *state* is the current state of the CIG system (C_FILE_XFER).

The following table summarizes the processing performed by file_control in response to each valid message type it receives from the Simulation Host. The first column lists the messages in alphabetical order. The second column briefly describes the purpose of the message (in italicized letters), then lists the major steps performed by file_control to process the message.

Message from SIM Host	Processing by file_control
MSG_CIG_CTL C_NULL C_STOP	*Causes a transition to another performance state.* No action. Returns to upstart.
MSG_DR11_PKT_SIZE	*Specifies exchange packet parameters.* Sets CIG and SIM exchange packet size, local terrain chunk size, and local terrain message interval.
MSG_END	*Signals end of packet buffer.* Calls EXCHANGE_DATA to send output and receive input buffers.
MSG_FILE_DESCR DB_CIG2SIM DB_SIM2CIG DB_DELETION DB_DIRECTORY DB_REN_FROM DB_REN_TO	*Transfers and manages files.* Uploads file from CIG to SIM; generates MSG_FILE_STATUS return message. Downloads file from SIM to CIG; generates MSG_FILE_STATUS return message. Deletes file from CIG disk; generates MSG_FILE_STATUS return message. Passes CIG directory data to SIM; generates MSG_FILE_STATUS return message. Finds file with this name; generates MSG_FILE_STATUS return message. Renames file to this name; generates MSG_FILE_STATUS return message.
MSG_FILE_STATUS	*Provides response for file transfer.* Resends block or aborts if message indicates.
MSG_FILE_XFER	*Contains the name of the file to upload or download.* Reads or writes data; generates MSG_FILE_STATUS return message.

Called By: upstart

Routines Called:
close
create_sz
EXCHANGE_DATA
lseek
open
printf
read
rsec
strcpy
strlen
SYSERR
system
write

Parameters: INT_2 state

Returns: none

2.2.3.11 find_fn.c

The find_fn function finds the file that has the highest extension and whose name matches a given character string. This ensures that the calling procedure loads the latest version of a file.

The function call is **find_fn(compare, n, exact, file_name)**, where:

compare is the name to be matched
n is the number of characters in the compare string
exact specifies whether or not the file name must match the compare string exactly
file_name is a pointer to the file name found by find_fn

The returned parameter (*success*) is set to 1 if a match is found, or -1 if no match is found.

Called By:	bootup_slave133 gsp_load	
Routines Called:	strcmp system	
Parameters:	char char INT_2 BOOLEAN	*compare *file_name n exact
Returns:	success	

2.2.3.12 fxbvtofl.c

The fxbvtofl CSU contains functions used to convert a fixed point bounding volume to floating point. These functions are:

- fxbvtofl
- fxbvtofl_dart
- fxbvtofl_020

2.2.3.12.1 fxbvtofl

The fxbvtofl function converts a fixed point bounding volume to floating point.

The function call is **fxbvtofl(tobv, frombv)**, where:

tobv is the floating point bvol entry
frombv is the fixed point bvol entry

This function is not currently used.

Called By:	none	
Routines Called:	FXTO881	
Parameters:	BVOL_ENTRY FIX_BVOL_ENTRY	*tobv *frombv
Returns:	none	

2.2.3.12.2 fxbvtofl_dart

The fxbvtofl_dart function converts a fixed point bounding volume to floating point.

The function call is **fxbvtofl_dart(tobv, frombv)**, where:

tobv is the floating point bvol entry
frombv is the fixed point bvol entry

This function is not currently used.

Called By:	none	
Routines Called:	FXTO881	
Parameters:	BVOL_ENTRY FIX_BVOL_ENTRY	*tobv *frombv
Returns:	none	

2.2.3.12.3 fxbvtofl_020

The fxbvtofl_020 function converts a fixed point bounding volume to floating point

The function call is **fxbvtofl_020(tobv, frombv)**, where:

tobv is the floating point bvol entry
frombv is the fixed point bvol entry

Called By:	download_bvols

Routines Called:	FXTO881	
Parameters:	BVOL_ENTRY FIX_BVOL_ENTRY	*tobv *frombv
Returns:	none	

2.2.3.13 gsp_load.c

The gsp_load function loads the Force and GSP (Graphics System Processor) boards with data and code for execution. gsp_load is called by db_mcc_setup if the system has a Force board and GSP has not yet been initialized.

The function call is **gsp_load(force_start)**, where *force_start* is TRUE if a Force board is present. gsp_load does the following:

- Initializes Force variables.
- Loads the latest version of the forcetask from disk.
- Starts the forcetask.
- Halts the GSP task.
- Runs a test on GSP memory.
- Loads the latest versions of the following GSP files from disk: bitmap, lookut (the color lookup table), data2d (the 2-D overlay database), and task2d (the GSP task).
- Starts the GSP task.

The Force and GSP boards are used to generate and display two-dimensional overlays on 120TX systems.

Called By:	db_mcc_setup	
Routines Called:	find_fn printf system TRIGGER_FORCE WAIT_FORCE XCLOSE XOPEN XREAD	
Parameters:	BOOLEAN	force_start
Returns:	none	

2.2.3.14 gun_overlays.c

The functions in gun_overlays.c are used to build M1 and M2 overlays. These overlays are hard-coded displays of three-dimensional polygons that are displayed on the viewport, over the terrain display. The overlay shows objects that would normally obscure the view of the terrain, to better emulate the real-world view out the vehicle's window. Overlays are vehicle-specific.

gun_overlays contains the following functions:

- m1_gun_overlay
- m2_gun_overlay
- make_m1_overlays
- make_m2_overlays

These functions apply to the 120T CIG only. Overlays on the 120TX are generated by the 2-D overlay compiler using Simulation Host messages.

2.2.3.14.1 m1_gun_overlay

The m1_gun_overlay function creates gun and gunner overlays for M1 vehicles. This function is called by simulation when the message from the Simulation Host is MSG_GUN_OVERLAY and the message type is M1_OVERLAYS.

The function call is **m1_gun_overlay(pmsg, db)**, where:

pmsg is a pointer to the MSG_GUN_OVERLAY message
db is the double-buffer memory current base pointer

Gun overlays show the components of the gun (on the simulation vehicle) that would be visible when looking out from the vehicle's window. Gunner overlays show cross-hairs and digits. The MSG_GUN_OVERLAY message specifies the digits to be displayed.

Called By:	simulation	
Routines Called:	none	
Parameters:	MSG_GUN_OVERLAY INT_4	*pmsg db
Returns:	none	

2.2.3.14.2 m2_gun_overlay

The m2_gun_overlay function creates gun overlays for M2 vehicles. This function is called by simulation when the message from the Simulation Host is MSG_GUN_OVERLAY and the message type is M2_OVERLAYS.

The function call is **m2_gun_overlay(pmsg, db)**, where:

pmsg is a pointer to the MSG_GUN_OVERLAY message
db is the double-buffer memory current base pointer

Gun overlays show the components of the gun (on the simulation vehicle) that would be visible when looking out from the vehicle's window. Gunner overlays show cross-hairs and digits. The MSG_GUN_OVERLAY message specifies the digits to be displayed.

Called By:	simulation	
Routines Called:	none	
Parameters:	MSG_GUN_OVERLAY INT_4	*pmsg db
Returns:	none	

2.2.3.14.3 make_m1_overlays

The make_m1_overlays function sets up M1 overlay data at viewport configuration time. This function is called by overlay_setup in the Viewport Configuration component if the Simulation Host sends a MSG_OVERLAY_SETUP message with the type set to 1 (M1_OVERLAYS).

Note: *The MSG_OVERLAY_SETUP message can specify gunners_viewport (the viewport that is to have the gunner's overlay) and barrel_viewports (the viewports the gun barrel is to be viewable in). These values are not currently used. The gunner's overlay is placed on any viewport belonging to a configuration node that has bit 0 of its branch mask set. The gun barrel overlay is placed on any viewport belonging to a configuration node that has bit 1 of its branch mask set.*

The function call is **make_m1_overlays (po, ppg)**, where:

po is a pointer to the overlay parameters
ppg is a pointer to the M1_GUN_OVERLAY message

Called By:	overlay_setup

Routines Called:	aam_malloc id_4x3mtx make_p_nt swap_axis	
Parameters:	OVERLAY_PARAMS M1_GUN_OVERLAY	*po **ppg
Returns:	none	

2.2.3.14.4 make_m2_overlays

The make_m2_overlays routine sets up M2 overlay data at viewport configuration time. This function is called by overlay_setup in the Viewport Configuration component if the Simulation Host sends a MSG_OVERLAY_SETUP message with the message type set to 2 (M2_OVERLAYS).

Note: The MSG_OVERLAY_SETUP message can specify gunners_viewport (the viewport that is to have the gunner's overlay) and barrel_viewports (the viewports the gun barrel is to be viewable in). These values are not currently used. The gunner's overlay is placed on any viewport belonging to a configuration node that has bit 0 of its branch mask set. The gun barrel overlay is placed on any viewport belonging to a configuration node that has bit 1 of its branch mask set.

The function call is **make_m2_overlays (po, ppg)**, where:

po is a pointer to the overlay parameters
ppg is a pointer to the M2_GUN_OVERLAY message

Called By:	overlay_setup	
Routines Called:	aam_malloc id_4x3mtx make_p_nt swap_axis	
Parameters:	OVERLAY_PARAMS M1_GUN_OVERLAY	*po **ppg
Returns:	none	

2.2.3.15 hw_test.c

The hw_test function processes messages from the SIM to handle hardware tests. hw_test is called by upstart whenever the state requested by the Simulation Host is C_TEST_MODE.

The function call is **hw_test(state)**, where *state* is the current state of the CIG system (C_TEST_MODE).

The following table summarizes the processing performed by hw_test in response to each valid message type it receives from the Simulation Host. The first column lists the messages in alphabetical order. The second column briefly describes the purpose of the message (in italics), then lists the major steps performed by hw_test to process the message.

Message from SIM Host	Processing by hw_test
MSG_CIG_CTL C_NULL C_STOP	*Causes a transition to another performance state.* No action. Returns to upstart.
MSG_DR11_PKT_SIZE	*Specifies exchange packet parameters.* Sets CIG and SIM exchange packet size, local terrain chunk size, and local terrain message interval.
MSG_END	*Signals end of packet buffer.* Calls EXCHANGE_DATA to send output and receive input buffers.
MSG_TEST_NAME ECHO_PKT	*Specifies test to be run.* Echoes packet back to SIM. (This test is not currently implemented.)

Called By: upstart

Routines Called: EXCHANGE_DATA
printf
SYSERR

Parameters: INT_2 state

Returns: none

2.2.3.16 load_dbase.c

The load_dbase function loads the terrain database into active area memory, and sets up various tables with the necessary data from the database. It also calls open_ded to load the contents of the dynamic elements database (DED). load_dbase is called by db_mcc_setup after the viewport configuration tree has been created.

The function call is **load_dbase(db_name, state)**, where:

db_name is the name of the database
state is the current state of the CIG system (C_DB_SETUP or C_MCC_SETUP)

load_dbase does the following:

- Determines how much generic memory is available.
- If not enough memory is available, truncates the number of bytes to what is available.
- Reads in the data from the specified database.
- Processes the model directory entries.
- Reads in the overflow terrain data, if there is sufficient room.
- Calls open_ded to open the dynamic elements database, read the models in, and process them.
- Calls load_modules to load the initial load modules.
- Initializes the Load Module Branch Table, subroutine call table, and field-of-view test table for a 3500-meter or 7000-meter viewing range.
- Sets the *database_is_open* flag to TRUE.

Called By:	db_mcc_setup	
Routines Called:	bus_error free load_modules malloc open_ded printf XLSEEK XREAD	
Parameters:	char INT_2	db_name[] state
Returns:	none	

2.2.3.17 make_bbn.c

The functions in make_bbn.c are used by gossip to make and modify hull-to-world matrices for debugging purposes. These functions are:

- prt_mtx
- rotate_x
- rotate_y
- rotate_z
- multmatrix

- id_matrix

These routines are used only by model_mtx, which is called by gos_model. They are invoked only if debug mode has been enabled.

The routines used to make and update matrices for the simulation are contained in the mkmtx_nt.c CSU.

2.2.3.17.1 prt_mtx

The prt_mtx function copies a matrix in memory.

The function call is **prt_mtx(matrix, pntr)**, where:

matrix is the matrix
pntr is a pointer to the destination memory location

Called By:	model_mtx	
Routines Called:	none	
Parameters:	REAL_4 REAL_4	matrix[4][3] *pntr
Returns:	none	

2.2.3.17.2 rotate_x

The rotate_x function rotates a matrix about the X axis.

The function call is **rotate_x(theta, matrix)**, where:

theta is the angle of rotation
matrix is the matrix to be rotated

Called By:	model_mtx	
Routines Called:	cos id_matrix multmatrix sin toradians	
Parameters:	REAL_4 REAL_4	theta matrix[4][3]

Returns: none

2.2.3.17.3 rotate_y

The rotate_y function rotates a matrix about the Y axis.

The function call is **rotate_y(theta, matrix)**, where:

theta is the angle of rotation
matrix is the matrix to be rotated

Called By:	model_mtx	
Routines Called:	cos id_matrix multmatrix sin toradians	
Parameters:	REAL_4 REAL_4	theta matrix[4][3]
Returns:	none	

2.2.3.17.4 rotate_z

The rotate_z function rotates a matrix about the Z axis.

The function call is **rotate_z(theta, matrix)**, where:

theta is the angle of rotation
matrix is the matrix to be rotated

Called By:	model_mtx	
Routines Called:	cos id_matrix multmatrix sin toradians	
Parameters:	REAL_4 REAL_4	theta matrix[4][3]

Returns:	none

2.2.3.17.5 multmatrix

The multmatrix function multiplies two matrices together. This function is used to multiply a matrix by a rotation matrix.

The function call is **multmatrix(matrix, matrix_tmp)**, where:

matrix is the rotation matrix
matrix_tmp is the matrix to be rotated

Called By:	rotate_x rotate_y rotate_z	
Routines Called:	none	
Parameters:	REAL_4 REAL_4	matrix[4][3] matrix_tmp[4][3]
Returns:	none	

2.2.3.17.6 id_matrix

The id_matrix function creates an identity matrix (positioned at the origin) for use in rotating matrices.

The function call is **id_matrix(matrix)**, where *matrix* is the identity matrix to be created.

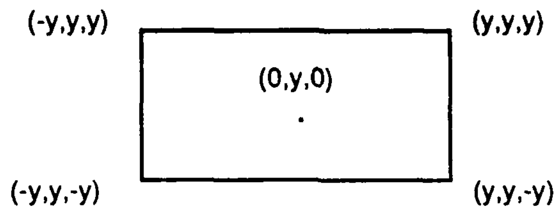
Called By:	model_mtx rotate_x rotate_y rotate_z	
Routines Called:	none	
Parameters:	REAL_4	matrix[4][3]
Returns:	none	

2.2.3.18 mkcal.c

The functions in mkcal.c generate monitor calibration images. These functions are:

- make_cal_overlay
- pix_mult

The Poly Processor uses perspective matrices in normalized viewspace (i.e., the field-of-view is not used) when crunching on overlay polygons. The only perspective matrix required for an overlay is a matrix to swap the axes (view space into screen space). The vertices overlay can be described to the Poly Processor as follows:



where y is the distance from the eye to the overlay.

This means that if the vertices of an overlay (such as the monitor calibration overlay) are given in pixel coordinates, they must be converted to the normalized view space coordinate system. For example, if the screen resolution is 200 x 200, a vertex with pixel coordinates (-50,100) is converted to (-1/2,1).

2.2.3.18.1 make_cal_overlay

The make_cal_overlay function allocates and makes a calibration overlay. This function is called by cig_config (in Viewport Configuration) as part of its initialization process.

The calibration overlay is a hard-coded pattern of triangles, vertical and horizontal alignment bars, and colored rectangles. The overlay is displayed on a viewport on top of the view of the terrain. The pattern helps the Simulator user center the screen.

The function call is **make_cal_overlay()**.

Called By:	cig_config
Routines Called:	aam_malloc id_4x3mtx swap_axis
Parameters:	none
Returns:	none

2.2.3.18.2 pix_mult

The pix_mult function converts pixel coordinates into normalized viewspace coordinates.

The function call is **pix_mult(resolution, y_dist)**, where:

resolution is the screen resolution
y_dist is the y pixel coordinate

The function divides *y_dist* by (*resolution* * .5) and returns the result as *mult*.

Called By:	none	
Routines Called:	none	
Parameters:	INT_2 REAL_4	resolution y_dist
Returns:	mult	

2.2.3.19 mkmtx_nt.c

The functions in mkmtx_nt.c are used to rotate and translate matrices. These functions are:

- make_p_nt
- rotate_x_nt
- rotate_y_nt
- rotate_z_nt
- swap_axis
- id_4x3mtx
- scale_mtx
- translate
- mult_4x3mtx
- getmatrix
- matrix2
- mtxcpy

2.2.3.19.1 make_p_nt

The make_p_nt function converts a matrix to a perspective 4x3 matrix.

The function call is **make_p_nt(itan_fov_i, itan_fov_j, hoffset_x, hoffset_y, matrix)**, where:

itan_fov_i is inverse of the tangent of the horizontal field-of-view angle
itan_fov_j is inverse of the tangent of the vertical field-of-view angle

hoffset_x is the horizontal offset of the x coordinate
hoffset_y is the horizontal offset of the y coordinate
matrix is the matrix to be converted

Called By:	make_m1_overlays make_m2_overlays viewspace_mtx	
Routines Called:	id_4x3mtx mult_4x3mtx	
Parameters:	REAL_4 REAL_4 REAL_4 REAL_4 REAL_4	itan_fov_i itan_fov_j hoffset_x hoffset_y matrix[4][3]
Returns:	none	

2.2.3.19.2 rotate_x_nt

The rotate_x_nt function rotates a 4x3 matrix about the X axis. This function is called by concat_mtx to change the pitch of an RTS3x3 (HPRXYZS) matrix.

The function call is **rotate_x_nt(cos_theta, sin_theta, matrix)**, where:

cos_theta is the cosine of the angle of rotation
sin_theta is the sine of the angle of rotation
matrix is the matrix to be rotated

Called By:	concat_mtx gos_model	
Routines Called:	id_4x3mtx mult_4x3mtx	
Parameters:	REAL_4 REAL_4 REAL_4	cos_theta sin_theta matrix[4][3]
Returns:	none	

2.2.3.19.3 rotate_y_nt

The rotate_y_nt function rotates a 4x3 matrix about the Y axis. This function is called by concat_mtx to change the roll of an RTS3x3 (HPRXYZS) matrix.

The function call is **rotate_y_nt(cos_theta, sin_theta, matrix)**, where:

cos_theta is the cosine of the angle of rotation
sin_theta is the sine of the angle of rotation
matrix is the matrix to be rotated

Called By:	concat_mtx gos_model	
Routines Called:	id_4x3mtx mult_4x3mtx	
Parameters:	REAL_4 REAL_4 REAL_4	cos_theta sin_theta matrix[4][3]
Returns:	none	

2.2.3.19.4 rotate_z_nt

The rotate_z_nt function rotates a 4x3 matrix about the Z axis. This function is called by concat_mtx to change the heading of an RTS3x3 (HPRXYZS) matrix.

The function call is **rotate_z_nt(cos_theta, sin_theta, matrix)**, where:

cos_theta is the cosine of the angle of rotation
sin_theta is the sine of the angle of rotation
matrix is the matrix to be rotated

Called By:	concat_mtx gos_model viewspace_mtx	
Routines Called:	id_4x3mtx mult_4x3mtx	
Parameters:	REAL_4 REAL_4 REAL_4	cos_theta sin_theta matrix[4][3]

Returns: none

2.2.3.19.5 swap_axis

The swap_axis function converts a matrix's axes so that the matrix conforms to the CIG's coordinate system, as follows:

xview = xworld
yview = -zworld
zview = yworld

The function call is **swap_axis(matrix)**, where *matrix* is the matrix to be converted. swap_axis first calls id_4x3mtx to create a 4x3 identity matrix. It then sets this matrix to the following:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

swap_axis then multiplies this matrix by the original matrix.

Called By: make_m1_overlays
make_m2_overlays
viewspace_mtx

Routines Called: id_4x3mtx
mult_4x3mtx

Parameters: REAL_4 matrix[4][3]

Returns: none

2.2.3.19.6 id_4x3mtx

The id_4x3mtx function creates a 4x3 identity matrix (positioned at the origin) for use in rotating matrices.

The function call is **id_4x3mtx(matrix)**, where *matrix* is the new identity matrix.

Called By: concat_mtx
make_m1_overlays
make_m2_overlays
viewspace_mtx

Routines Called:	none	
Parameters:	REAL_4	matrix[4][3]
Returns:	none	

2.2.3.19.7 scale_mtx

The scale_mtx function scales (enlarges, reduces, or skews) a 4x3 matrix. This function is used to adjust matrices if load module blocking is enabled. It is called by concat_mtx to change the scale of an RTS3x3 (HPRXYZS) matrix.

The function call is **scale_mtx(scale, matrix)**, where:

scale is the scaling factor
matrix is the matrix to be scaled

Called By:	concat_mtx viewspace_mtx	
Routines Called:	id_4x3mtx mult_4x3mtx	
Parameters:	REAL_4 REAL_4	matrix[4][3] scale[3]
Returns:	none	

2.2.3.19.8 translate

The translate function moves a matrix to a new position by adding a translation value to each of its coordinates. This function is called by concat_mtx to change the translation of an RTS3x3 (HPRXYZS) matrix.

The function call is **translate(xval, yval, zval, matrix)**, where:

xval is the amount to be added to the x coordinate
yval is the amount to be added to the y coordinate
zval is the amount to be added to the z coordinate
matrix is the matrix to be translated

Translation amounts are specified in meters.

Called By:	concat_mtx

Routines Called:	id_4x3mtx mult_4x3mtx	
Parameters:	REAL_4 REAL_4 REAL_4 REAL_4	xval yval zval matrix[4][3]
Returns:	none	

2.2.3.19.9 mult_4x3mtx

The mult_4x3mtx function multiplies two 4x3 matrices together. This function is used to multiply a matrix by a rotation matrix.

The function call is **mult_4x3mtx(matrix, matrix_tmp)**, where:

matrix is the rotation matrix
matrix_tmp is the matrix to be rotated

Called By:	concat_mtx make_p_nt rotate_x_nt rotate_y_nt rotate_z_nt scale_mtx swap_axis translate	
Routines Called:	none	
Parameters:	REAL_4 REAL_4	matrix[4][3] matrix_tmp[4][3]
Returns:	none	

2.2.3.19.10 getmatrix

The getmatrix function concatenates a matrix with matrix_tmp.

The function call is **getmatrix(matrix, matrix_tmp)**, where:

matrix is the original matrix and the result matrix
matrix_tmp is matrix to concatenate with the original matrix

Called By:	concat_mtx viewspace_mtx	
Routines Called:	none	
Parameters:	REAL_4 REAL_4	matrix[4][3] matrix_tmp[4][3]
Returns:	none	

2.2.3.19.11 matrix2

The matrix2 function concatenates (multiplies) two matrices to create a third matrix.

The function call is **matrix2(matrixa, matrixb, matrixc)**, where:

matrixa and *matrixb* are the matrices to be concatenated
matrixc is the result

This function is not currently used.

Called By:	none	
Routines Called:	none	
Parameters:	REAL_4 REAL_4 REAL_4	matrixa [4][3] matrixb [4][3] matrixc [4][3]
Returns:	none	

2.2.3.19.12 mtxcpy

The mtxcpy function copies a matrix from one memory location to another.

The function call is **mtxcpy(to_matrix, from_matrix, matrix_type)**, where:

to_matrix is the destination location
from_matrix is the source location
matrix_type is the type of matrix (RTS3x3_TYPE, RTS4x3_TYPE, or ROT2x1_TYPE)

Called By:	concat_mtx confignode_setup simulation viewport_setup	
Routines Called:	none	
Parameters:	I4P I4P BYTE	to_matrix from_matrix matrix_type
Returns:	none	

2.2.3.20 model_mtx.c

The model_mtx function builds hull-to-world, turret-to-hull, and gun-to-turret matrices. This function is called by gos_model for options that are available to the Gossip user only in debug mode.

The function call is **model_mtx(modnum)**, where *modnum* is the model number.

Called By:	gos_model	
Routines Called:	id_matrix prt_mtx rotate_x rotate_y rotate_z translate	
Parameters:	INT_2	modnum
Returns:	none	

2.2.3.21 open_dbase.c

The open_dbase function opens the terrain database and initializes configuration and active area memory parameters for Ballistics. open_dbase is called by db_mcc_setup when it receives a MSG_FILE_DESCR - DB_SETUP message.

The function call is **open_dbase(db_name, state)**, where:

db_name is the name of the database to be opened
state is the current state of the CIG system (C_DB_SETUP or C_MCC_SETUP)

open_dbase does the following:

- Opens the database file specified in the Simulation Host message or entered through the keyboard. Calls find_fn to find the latest version of the specified file.
- Reads the file header.
- Verifies that the database is compatible with the software.
- Initializes database variables: number of load module blocks per side, grid space, number of load modules on a side, number of load modules per side of a load module block, load module width, load module block width, active area width, total number of load modules and load module blocks, etc.
- Clears extra memory if load module blocking is enabled.
- Initializes Ballistics configuration parameters: processor type, frame rate, number of AAM partitions, maximum chord length, maximum model radius, maximum number of models, maximum number of active rounds, polygons, and bvols, etc.
- Sends the configuration data to Ballistics by pushing a MSG_B0_BAL_CONFIG message onto the Ballistics message queue.
- Initializes AAM partition information for Ballistics: number of load modules per side, total number of load modules in AAM, viewing distance, grid width, AAM base address, etc.
- Sends the AAM partition parameters to Ballistics by pushing a MSG_B0_DATABASE_INFO message onto the Ballistics message queue.

The terrain database is loaded into active area memory by load_dbase, which is called by db_mcc_setup after the viewport configuration tree is created.

Called By:	db_mcc_setup	
Routines Called:	find_fn free malloc mx_push printf strlen SYSERR XCLOSE XLSEEK XOPEN XREAD	
Parameters:	char INT_2	db_name[] state
Returns:	none	

2.2.3.22 open_ded.c

The open_ded function opens the dynamic elements database (DED) and processes the dynamic model list, changing the relative AAM addresses to absolute AAM addresses. open_ded is called by load_dbase after it loads the terrain database into active area memory.

The function call is **open_ded(ded_db_name, ded_start_address, avail_gm)**, where:

ded_db_name is the name of the dynamic elements database
ded_start_address is the location at which to start loading dynamic models
avail_gm is the amount of space in generic memory for model information

open_ded does the following:

- Finds the DED file. The file name is specified by the Simulation Host in the MSG_FILE_DESCR - DB_DED_SETUP message. db_mcc_setup sets the name (*ded_db_name*) in global memory, and load_dbase passes it to open_ded. The file name can also be specified through the keyboard. open_ded calls find_fn to find the latest version of the specified file.
- Opens the file.
- Reads the database header and verifies it is valid.
- Allocates memory for the model address, model catalog, special effects address, and special effects catalog tables.
- Verifies there is enough generic memory for the DED models.
- Loads the models into the generic model AAM.
- Calls download_bvols to download the models and bounding volumes to Ballistics.
- Processes the model directory entries.
- Processes the special effect directory entries.
- Closes the DED database file.

The function returns 0 if the DED is fully or partially loaded. It returns -1 if no DED databases are found.

Called By:	load_dbase	
Routines Called:	ded_model_trace download_bvols find_fn free malloc printf strlen XCLOSE XLSEEK XOPEN XREAD	
Parameters:	char INT_4 INT_4	ded_db_name[] ded_start_address avail_gm
Returns.	0 -1	

2.2.3.23 simulation.c

The simulation function is the message handler for the real-time simulation control of the CIG hardware and communications with the Simulation Host. simulation is called by db_mcc_setup when it receives a MSG_CIG_CTL message with the state set to C_MCC_SIMUL or C_SIMULATION.

The function call is **simulation(state, top_of_configtree)**, where:

> *state* is the current state of the CIG system (C_SIMULATION or C_MCC_SIMUL)
> *top_of_configtree* is a pointer to the root configuration node

simulation does the following:

- Initializes various static variables (round fired estimated impact time and range, southwest corner of AAM, static vehicle counter, etc.).
- Displays the coordinates of the northwest corner of the terrain database.
- Posts a message to the MONITOR_MB mailbox.
- Puts Ballistics into the run state:
  - Sets the Ballistics state to BX_RUN.
  - Pushes a MSG_B0_STATE_CONTROL message onto the Ballistics message queue.
- Sets the coordinates of the southwest corner of active area memory, based on the simulated vehicle's starting position.
- Tells Ballistics where AAM is by pushing a MSG_B0_AAM_SW_CORNER message onto the Ballistics message queue.
- Initializes the multiple-frame effects pointers to the field-of-view test table (for a 7000 meter viewing range) or the terrain (for a 3500-meter viewing range).
- Posts a message to the DATABASE_MB mailbox and waits for rowcol_rd to finish. rowcol_rd loads the initial load modules into active area memory.
- Posts a message to the LOCAL_TERRAIN_MB mailbox and waits for local_terrain to finish. local_terrain generates a message describing the terrain around the simulated vehicle for the Simulation Host.
- Initializes the local terrain message counter. This counter is used in conjunction with the local terrain interval to determine when to generate local terrain messages (currently set at every 32 frames).
- Determines the frame rate (15 or 30 Hz) and sets it in global memory.
- Tells Ballistics the frame rate by pushing a MSG_B0_CIG_FRAME_RATE message onto the Ballistics message queue.
- Determines which double buffer is being used by the hardware.
- Processes each runtime message received from the Simulation Host in turn (see table below).
- Reads and processes all hit, miss, and round position messages returned by Ballistics (from the Ballistics message queue).
- Processes laser return messages returned from Ballistics.
- Returns all messages passed back from the 2-D overlay processor.
- Performs AGL (above ground level) processing if enabled.
- Calls EXCHANGE DATA to exchange message packets.
- Resets the state tables and waits for the next interrupt.

The following table summarizes the processing performed by simulation in response to each valid message type it receives from the Simulation Host. The first column lists the messages in alphabetical order. The second column briefly describes the purpose of the message (in italics), then lists the major steps performed by simulation to process the message.

Message from SIM Host	Processing by simulation
MSG_1ROTATION	*Updates a single rotation of an hprxyzs matrix.* Changes heading, pitch, or roll as indicated; calls concat_mtx.
MSG_3ROTATIONS	*Updates the rotation portion (h,p,r) of an hprxyzs matrix.* Changes heading, pitch, and roll; calls concat_mtx.
MSG_AGL_SETUP	*Toggles AGL processing on and off.* Sets agl_wanted in global memory.
MSG_AMMO_DEFINE	*Define ammunition maps.* Sets ammo_map in global memory.
MSG_CIG_CTL C_NULL C_STOP	*Causes a transition to another performance state.* No action. Resets Ballistics; turns off monitors; initializes AAM; closes database; frees model and effect tables; returns to db_mcc_setup.
MSG_DR11_PKT_SIZE	*Specifies exchange packet parameters.* Sets CIG and SIM exchange packet size, local terrain chunk size, and local terrain message interval.
MSG_END	*Signals end of packet buffer.* Signals T&C board; processes changes to static vehicles; processes special effects; adds dynamic vehicles; tells Force board to transfer data to 2-D; counts down multiple frame effects; processes agl_wanted; sends new frame information to Ballistics; moves load module STP to quad buffer; waits for next interrupt.
MSG_GUN_OVERLAY	*Changes gun/gunner overlays.* Calls m1_gun_overlay or m2_gun_overlay, as appropriate.
MSG_HPRXYZS_MATRIX	*Updates a configuration node's matrix.* Calls mtxcpy; calls concat_mtx; calls process_vppos if a world/hull matrix.
MSG_OTHERVEH_STATE	*Describes the state of all dynamic vehicles in the terrain.* Puts vehicle's matrix data in model table; adds model to proper load module.
MSG_PASS_ON	*Tells simulation to pass the message on to a specific subsystem (2-D overlay processor).* Writes message data to Force memory.
MSG_PROCESS_ROUND	*Tells Ballistics to process a round.* Pushes MSG_B0_PROCESS_ROUND message onto Ballistics message queue.
MSG_REQUEST_LASER_- RANGE	*Asks for pixel depth for i, j position on screen.* Gets data from Force.
MSG_ROT2x1_MATRIX	*Updates a configuration node's matrix.* Calls concat_mtx.

MSG_ROUND_FIRED	*Tells Ballistics that a round has been fired.* Pushes MSG_B0_ROUND_FIRED message onto Ballistics message queue.
MSG_RTN_LT	*Requests a local terrain message; used only by the MCC station (state=C_MCC_SIMUL).* Posts message to invoke rowcol_rd; posts message to invoke local_terrain.
MSG_RTS4x3_MATRIX	*Updates a configuration node's matrix.* Calls concat_mtx; calls process_vppos if world/hull matrix node.
MSG_SCALE	*Updates the scale portion (x,y,z) of an hprxyzs matrix.* Unpacks coordinates from SIM Host; calls concat_mtx.
MSG_SHOW_EFFECT	*Used to show the effect of an impact on terrain or a vehicle.* Sets frame count for effect and adds to multi-frame effects list; adds effect to special effects table; finds load module the model is in.
MSG_STATICVEH_REM	*Removes a static vehicle from the local terrain.* Finds vehicle's load module; deletes vehicle from model table; pushes MSG_B0_DELETE_STATIC_VEHICLE message onto Ballistics message queue; generates error if vehicle out of viewing range.
MSG_STATICVEH_STATE	*Adds a static vehicle to the local terrain.* Increments count of static vehicles; updates model table; adds model to proper load module, pushes MSG_B0_ADD_-STATIC_VEHICLE message onto Ballistics message queue.
MSG_TRAJ_CHORD	*Used for chords that represent trajectories.* Pushes MSG_B0_TRAJ_CHORD message onto Ballistics message queue; for tracer messages, stores effect data in memory.
MSG_TRANSLATION	*Updates the translation portion (x,y,z) of an hprxyzs matrix.* Unpacks coordinates from SIM Host; calls concat_mtx; calls process_vppos if world/hull matrix.
MSG_VIEW_FLAGS	*Updates system view flags (e.g., on/off, FLIR, DTV) or branch values.* Calls process_vflags.
MSG_VIEW_MAGNIFICATION	*Changes viewport's field of-view and/or level of detail.* Calls update_fov.
MSG_VIEW_MODE	*Updates view mode (off, night, day, BW, WHT, BHT).* Sets calibration modifier; sets timing_control_word; loads AAM with view mode codes for DTP.

Called By: db_mcc_setup

Routines Called:
active_area_init
concat_mtx
EXCHANGE_DATA_SIM
FIND_LM
free
FXTOFL
m1_gun_overlay

m2_gun_overlay
mtxcpy
mx_peek
mx_push
mx_skip
printf
process_vflags
process_vppos
read_watch
return_aam_ptr
sc_accept
sc_pend
sc_post
start_watch
stop_watch
SYSERR
sysrup_off
sysrup_on
update_fov
XCLOSE

Parameters:	INT_2	state
	CONFIGURATION_NODE	*top_of_configtree

Returns: none

2.2.3.24 stdio.c

The stdio function is required for the OASYS compiler only. It defines stdin, stdout, and stderr.

This function is not currently used.

Called By: none

Routines Called: none

Parameters: none

Returns: none

2.2.3.25 support.c

The functions in support.c are Butterfly-compatible versions of some of the operating system service calls used by the real-time software. These functions are as follows:

- start_watch
- read_watch
- stop_watch
- bus_error
- bus_error_w
- system
- sload
- get_record
- send_data
- ver_data
- check_sum
- get_binary_data
- get_char
- ctoi
- unbf_getchar
- sysrup_on
- sysrup_off

2.2.3.25.1 start_watch

The start_watch function is a null stub for Butterfly compatibility. It is not currently used.

2.2.3.25.2 read_watch

The read_watch function is a null stub for Butterfly compatibility. It is not currently used.

2.2.3.25.3 stop_watch

The stop_watch function is a null stub for Butterfly compatibility. It is not currently used.

2.2.3.25.4 bus_error

The bus_error function is a Butterfly routine used to test whether a specified memory location exists.

The function call is **bus_error(address, accesstype)**, where:

address is the test address
accesstype is **b** (byte access), **w** (word access), or **l** (long word access)

bus_error returns *ret* set to 0 if the location exists, or 1 if it does not.

Called By:	main (in upstart)	
Routines Called:	restoreker	
Parameters:	INT	address

	char	accesstype
Returns:	ret	

2.2.3.25.5 bus_error_w

The bus_error_w function is a Butterfly routine used to test whether a specified memory location exists, and to write to that address.

The function call is **bus_error_w(address, accesstype, data)**, where:

address is the test address
accesstype is **b** (byte access), **w** (word access), or **l** (long word access)
data is the data to be written to the test address

bus_error_w returns *ret* set to 0 if the location exists, or 1 if it does not.

Called By:	main (in upstart)	
Routines Called:	restoreker	
Parameters:	INT char INT	address accesstype data
Returns:	ret	

2.2.3.25.6 system

The system function is a Butterfly routine used to execute a shell command.

The function call is **system(request, dat1, dat2, dat3)**, where:

request is the command to be executed: **20** (get root) or **24** (run file)
dat1 is the name of the file
dat2 is not used
dat3 is the offset for sload

The value returned (*ret*) is the size of the root directory or the value returned from sload.

Called By:	none
Routines Called:	bcopy Find_Value Map_Obj

printf
sload
Unmap_Obj

Parameters:	INT	request
	char	*dat1
	char	*dat2
	char	*dat3

Returns: ret

2.2.3.25.7 sload

The sload function converts a Motorola S-format file into executable code. It reads data from the disk in sector-sized chunks, breaks the ASCII down into record-sized lines, then stores the binary data.

The function call is **sload(filename, offset, wsize)**, where:

filename is the file to be converted
offset is the amount to add to the binary data address
wsize is the size of the destination granularity

The function returns 1 if successful, or -1 if it encounters an error (file could not be opened, bad checksum on a record, or early end-of-file detected).

Called By: system

Routines Called:
check_sum
get_binary_data
get_record
printf
send_data
ver_data
XCLOSE
XOPEN

Parameters:	char	*filename
	INT_4	offset
	char	wsize

Returns:
1
-1

2.2.3.25.8 get_record

The get_record function fills a string buffer with exactly one Motorola S-format record.

The function call is **get_record(record)**, where *record* is the record to be read.

The function returns the S-format byte count if successful. It returns 0 if there are no records in the file.

Called By:	sload	
Routines Called:	get_char	
Parameters:	BYTE	record[]
Returns:	0 byte_count	

2.2.3.25.9 send_data

The send_data function writes data to memory in ascending bytes from a given start address.

The function call is **send_data(address, cptr, count, wsize)**, where:

address is the initial load address (absolute S-format)
cptr is a pointer to the ASCII record characters
count is the number of characters to transmit
wsize is the size of the destination granularity

Called By:	sload	
Routines Called:	get_binary_data printf putchar	
Parameters:	WORD char INT_4 char	address *cptr count wsize
Returns:	none	

2.2.3.25.10 ver_data

The ver_data function compares ASCII characters with memory in ascending bytes from a given start address.

The function call is **ver_data(address, cptr, count)**, where:

address is the initial load address (absolute S-format)
cptr is a pointer to the ASCII record characters
count is the number of characters to compare

Called By:	sload	
Routines Called:	get_binary_data printf	
Parameters:	WORD char INT_4	address *cptr count
Returns:	none	

2.2.3.25.11 check_sum

The check_sum function verifies the checksum byte of an S-format record.

The function call is **check_sum(pointer, count)**, where:

pointer points to the record to be checksummed
count is the byte count

The *answer* returned by the function is 0 if the checksum byte is correct. A non-zero value indicates a bad checksum.

Called By:	sload	
Routines Called:	get_binary_data	
Parameters:	char INT_4	*pointer count
Returns:	answer	

2.2.3.25.12 get_binary_data

The get_binary_data function returns the binary equivalent of specified characters.

The function call is **get_binary_data(cptr, count)**, where:

cptr is a pointer to the character string
count is the number of characters to be converted

The result is returned as *binary_data*.

Called By:	check_sum get_record send_data sload ver_data	
Routines Called:	ctoi	
Parameters:	char INT_4	*cptr count
Returns:	binary_data	

2.2.3.25.13 get_char

The get_char function returns the next available ASCII character from a sector-sized buffer. If a character is found, get_char returns the integer. If the buffer is empty, get_char reads the next sector from disk. If there is no next sector, get_char returns EOF.

The function call is **get_char()**.

Called By:	get_record unbf_getchar
Routines Called:	fflush printf XREAD
Parameters:	none
Returns:	*bptr++ EOF

2.2.3.25.14 ctoi

The ctoi function converts a character to an integer.

The function call is **ctoi(c)**, where *c* is the character to be converted.

Called By:	get_binary_data	
Routines Called:	none	
Parameters:	char	c
Returns:	c - '0' c - 'A' + 10	

2.2.3.25.15 unbf_getchar

The unbf_getchar function is a Butterfly routine that gets a single character input from the standard input non-blocking I/O.

The function call is **unbf_getchar()**. The character is returned as *c*.

Called By:	none
Routines Called:	fflush get_char printf
Parameters:	none
Returns:	c

2.2.3.25.16 sysrup_on

The sysrup_on function is a null stub for Butterfly compatibility. It is not currently used.

2.2.3.25.17 sysrup_off

The sysrup_off function is a null stub for Butterfly compatibility. It is not currently used.

2.2.3.26 upstart.c

The upstart.c CSU contains the functions that form the driver for the real-time applications software. These functions are the following:

- main (for Butterfly compatibility only)
- templates_init (for Butterfly compatibility only)
- upstart
- bootup_slave133

2.2.3.26.1 main

The main function is used to start upstart. This function is provided for Butterfly compatibility only. It remaps the required addresses to VME addresses, then calls upstart.

main requires three arguments to start upstart: *host_id*, *my_id*, and *bvme_id*.

Called By:	none	
Routines Called:	atoi bus_error bzero Find_Value Make_Event Make_Obj map_vme Name_Bind printf remap_vme upstart	
Parameters:	int char	argc *argv[]
Returns:	none	

2.2.3.26.2 templates_init

The templates_init function initializes the data used to build the AAM data structures locally before copying them into the AAM.. This function is required for Butterfly compatibility only.

The function call is **templates_init()**.

Called By:	upstart

Routines Called:	bcopy labs_dgi_buffers_init
Parameters:	none
Returns:	none

2.2.3.26.3 upstart

The upstart function is the driver for the real-time applications software. It establishes communication with the Simulation Host, reads a message, then calls the appropriate function depending on the system state requested in the message.

upstart is initiated by rtt during the task initialization state. It does the following:

- Locates the T&C (Timing and Control) board.
- Loads Ballistics from disk.
- Posts a BALLISTICS_MB mailbox message to start Ballistics.
- Calls bootup_slave133 if a slave board is detected.
- Waits for Ballistics to return a status message and a global address message.
- Initializes the DR11 buffer sizes.
- Initializes the local terrain chunk size and the interval between local terrain messages.
- Initializes the system tasks.
- Calls OPEN_EXCHANGE to open the necessary pipes to the Simulation Host.
- Initializes active area memory.
- Processes messages from the Simulation Host, calling other functions as required.

The following table summarizes the processing performed by upstart in response to each valid message type it receives from the Simulation Host. The first column lists the messages in alphabetical order. The second column briefly describes the purpose of the message (in italics), then lists the major steps upstart performs to process the message.

Message from SIM Host	Processing by upstart
MSG_CIG_CTL C_DB_SETUP C_FILE_XFER C_MCC_SETUP C_NULL C_STOP C_TEST_MODE	*Causes a transition to another performance state.* Calls db_mcc_setup with state set to C_DB_SETUP. Calls file_control with state set to C_FILE_XFER. Calls db_mcc_setup with state set to C_MCC_SETUP. No action. No action. Calls hw_test with state set to C_TEST_MODE.
MSG_DR11_PKT_SIZE	*Specifies exchange packet parameters.* Sets CIG and SIM exchange packet size, local terrain chunk size, and local terrain message interval.
MSG_END	*Signals end of packet buffer.* Calls EXCHANGE_DATA (with state set to C_STOP) to send output and receive input buffers.

Called By: none

Routines Called:
active_area_init
bootup_slave133
bus_error
db_mcc_setup
EXCHANGE_DATA
file_control
hw_test
labs_dgi_buffers_init
malloc
mx_error
mx_open
mx_peek
mx_skip
OPEN_EXCHANGE
printf
sc_post
sin
SYSERR
templates_init (Butterfly only)
TORAD

Parameters: none

Returns: none

2.2.3.26.4 bootup_slave133

The bootup_slave133 function boots up the slave 133 board. The function first checks to see if the Ballistics file has already been loaded. If not, it loads the latest version of the Ballistics file from disk. If no Ballistics task is found on disk, the function resets the Ballistics board type to master.

The function call is **bootup_slave133()**.

Called By:	upstart
Routines Called:	find_fn printf strcpy system
Parameters:	none
Returns:	none

2.2.4 2-D Overlay Compiler [120TX systems only]

This section describes the functions that make up the 2-D (Two-Dimensional) Overlay Compiler, which is a major functional component of the CIG Host Mainline (UPSTART) CSC. These functions build the 2-D overlays from ASCII commands, then generate executable commands for the 2-D processor.

> *Note: These functions apply to 120TX systems only. The only overlays available on 120T systems are the hard-coded gun, gunner, and calibration overlays generated in the Real-Time Processing component.*

2-D overlays are displayed on a viewport on top of the three-dimensional terrain display. For example, overlays can be used to display calibration patterns and numerical readouts such as current altitude and speed. Each 2-D component is classified as either dynamic (able to move or change) or static (not capable of movement or change).

The 2-D overlay database describes all components that can be displayed in the overlays. This database is an ASCII file sent from the Simulation Host via messages. The overall process for creating the 2-D overlay database is as follows:

1. The Simulation Host invokes the 2-D compiler using the CIG Control - Start 2D Setup message.
2. The Simulation Host sends the ASCII file via 2-D SETUP messages, one per packet buffer.
3. After the entire file has been sent, the Simulation Host sends a CIG Control - Stop message.
4. The 2-D compiler function compiles the data. If a monitor is available, error and status information is displayed.
5. The data is downloaded via the Force board into 2-D dynamic memory on the GSP (Graphics System Processor) chip on the MPV board.

Once the 2-D database is loaded into memory, the overlays can be changed using PASS_ON messages sent from the Simulation Host. These messages contain commands that are used to move or change dynamic components, and to draw or remove static components. The 2-D task (which runs on the GSP) decodes the runtime commands and updates the component information in the 2-D database accordingly. The 2-D task then processes the changes to each component in the order in which they are defined in the database.

The functions in the 2-D Overlay Compiler CSU are not involved with runtime changes. The commands are passed directly from the real-time software to Force to the GSP, and the GSP processes the changes to the structures in its memory.

For the complete syntax of each command used to set up or change a 2-D image, refer to the "2-D Commands and Parameters" document. That document also provides a sample 2-D overlay and its ASCII input file.

Overlays can also be created and compiled offline. A special version of the 2-D compiler function is used to read the overlay file and generate a binary file. This file can then be copied to the CIG and downloaded to 2-D memory (via the Force board) at a later time. All of the source files that contain the functions used to process an overlay file offline are prefixed by u_; these functions are not described in this document A separate "make" file is used at system build time to select these source files instead of their online equivalents.

The primary data structures built by the 2-D compiler are the following:

Component descriptor table
Contains each component's number (0-63), color, channel (0 for high resolution, 1 or 2 for low-resolution), plane (foreground or background), window number (0 for screen space, 1-15 for user-defined windows), clipping values, pre-translate (pre-rotation) values, and post-translate (post-rotation) values.

Window descriptor table
Contains each window's absolute address, width (horizontal pixels), height (vertical pixels), pitch, and a conversion factor for GSP.

Component pointer table
Contains a pointer to each component in the 2-D database.

After compilation, these structures are downloaded into GSP memory. If the 2-D compiler is being run off-line, the data is compiled into a binary file which can later be downloaded to the GSP. Figure 2-7 illustrates these structures, their contents, and their inter-relationships, as they exist in GSP memory.

The primitive types handled by the 2-D compiler, and the functions used to process them, are the following:

Primitive	2-D Setup Function
bit_blt	setup_bit_blt
draw_line	setup_draw_line
draw_oval	setup_oval_rectangle
draw_rect	setup_oval_rectangle
fill_oval	setup_oval_rectangle
fill_poly	setup_poly
fill_rect	setup_oval_rectangle
polyline	setup_poly
string	setup_define_string
text	setup_text

The specified function is responsible for retrieving the parameters associated with the primitive, validating the data, then adding the data to the component descriptor table.

The structure of each of these primitives is illustrated in Figure 2-8.

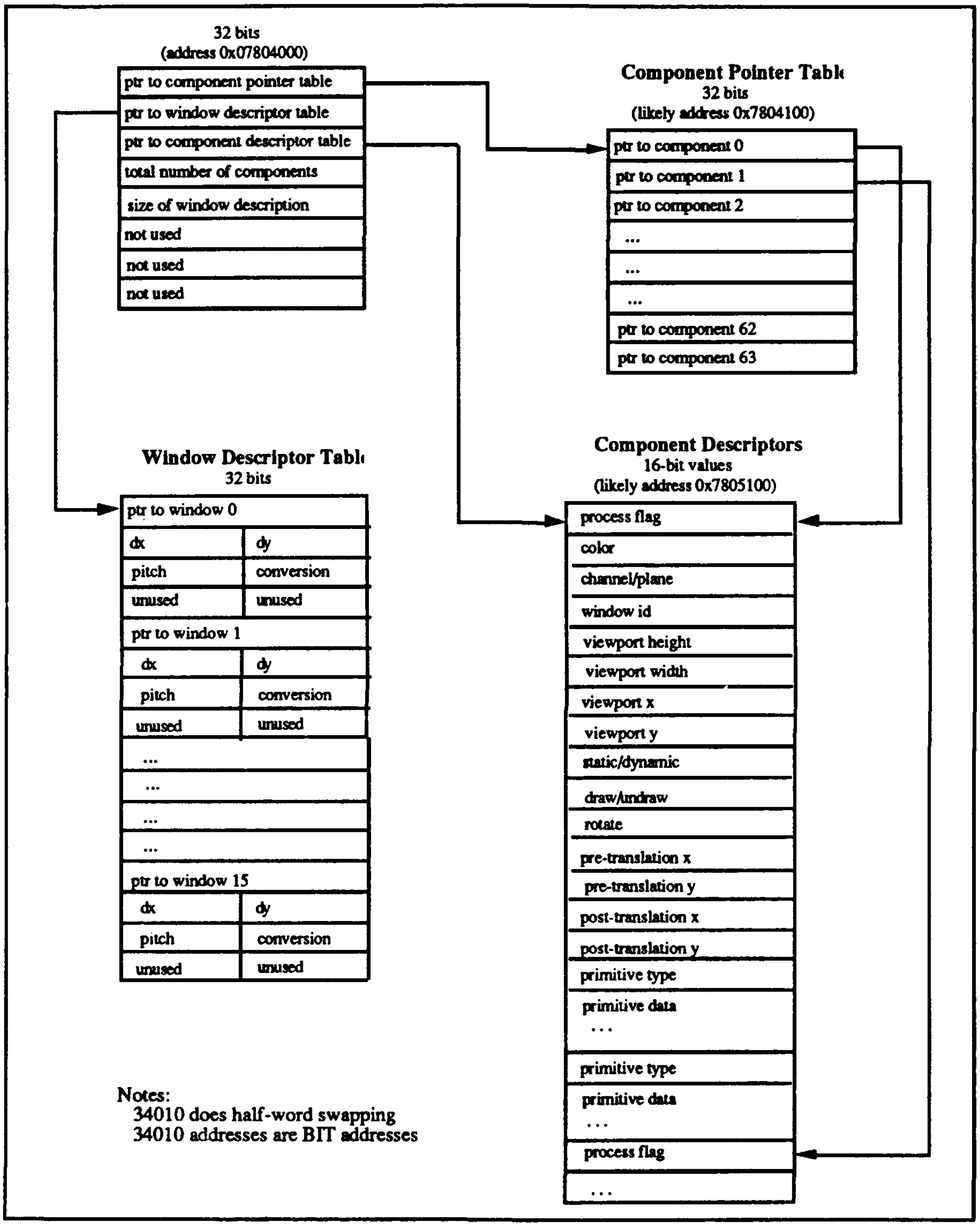


Figure 2-7. 2-D Memory (From The 2-D Compiler)

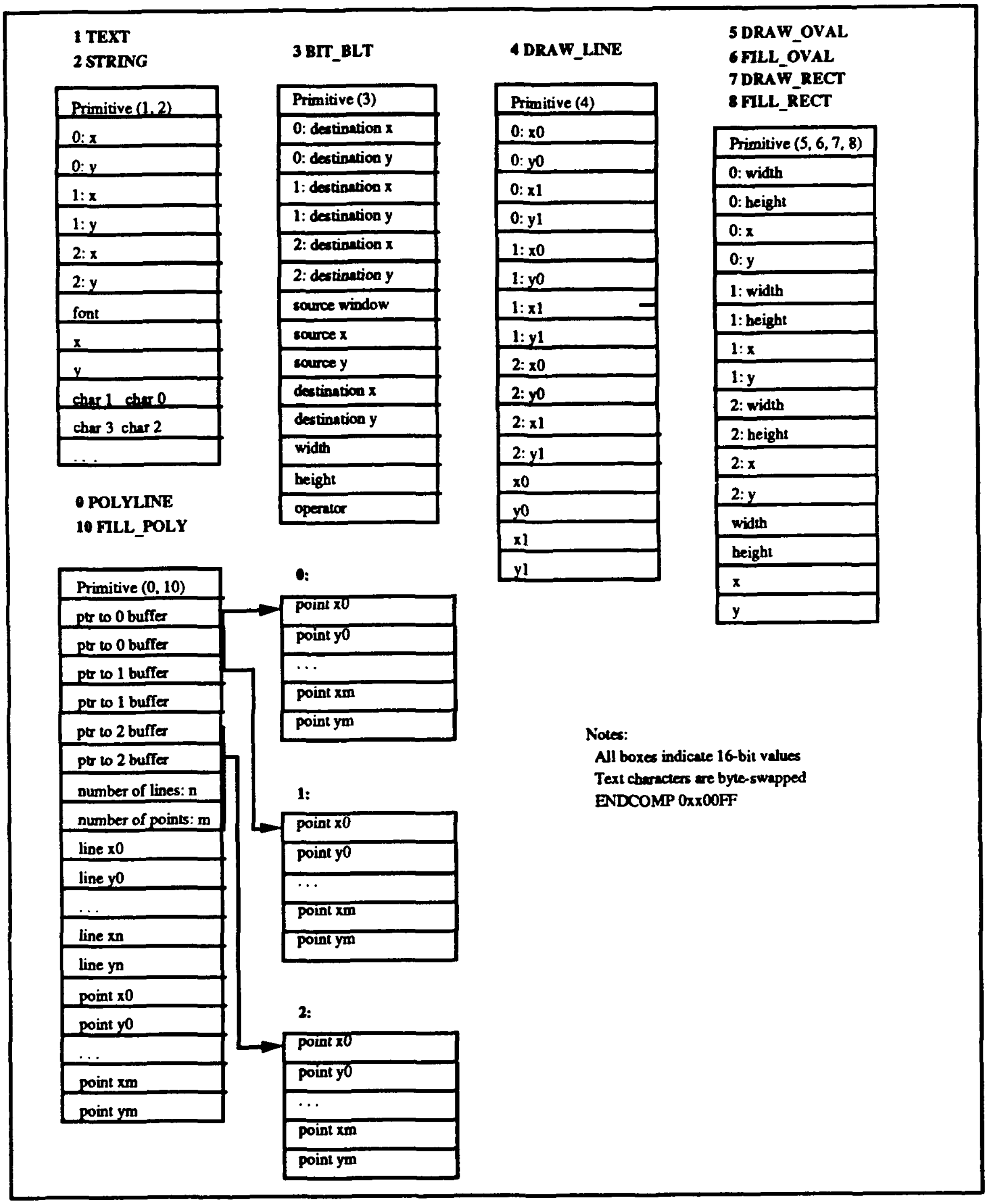


Figure 2-8. 2-D Compiler Primitives

Figure 2-9 identifies the CSUs in the 2-D Overlay Compiler component of the UPSTART CSC. These CSUs are described in this section.

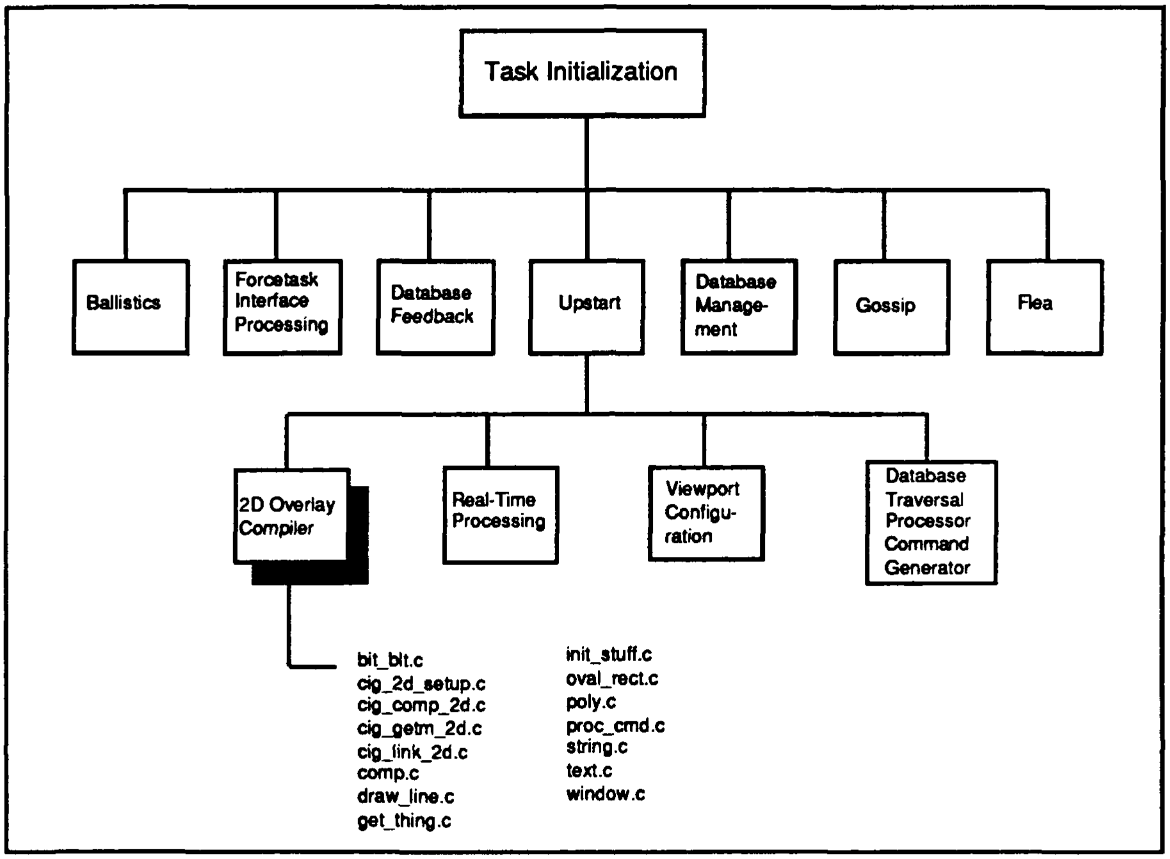


Figure 2-9. 2-D Overlay Compiler CSUs

Figure 2-10 illustrates how the CSUs in the 2-D Overlay Compiler interact.

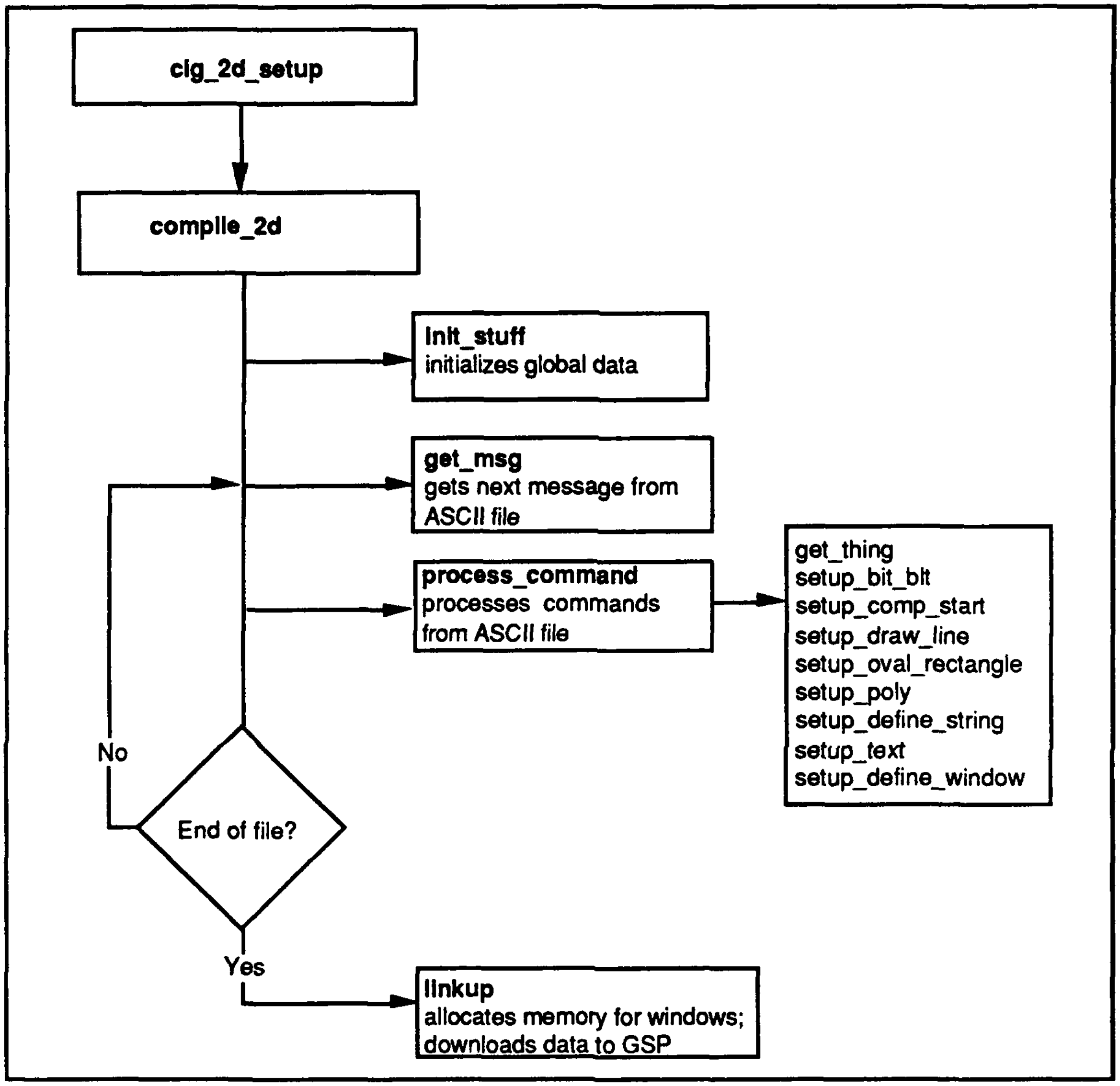


Figure 2-10. 2-D Overlay Compiler Flow Diagram

2.2.4.1 bit_blt.c (setup_bit_blt)

The bit_blt.c CSU contains one function, setup_bit_blt. This function is responsible for setting up block-transferring pixel information in the component descriptor table.

The function call is **setup_bit_blt(cmd)**, where *cmd* is the command (N_BIT_BLT) passed by process_command.

setup_bit_blt does the following:

- Verifies that component start data has already been processed.

- Calls get_thing to retrieve the parameters associated with this primitive.
- Determines if the component descriptor table has room available.
- Places the source window pixel x and y into the component descriptor table.
- Places the destination window pixel x and y into the component descriptor table.
- Places the width, height, and operator into the component descriptor table.

The *rtn_val* returned by the function is one of the following:

- SUCCESS if the data is added to the table successfully.
- COMPONENT_DESCRIPTOR_TBL_FULL if the table does not have enough room for the new data.
- SYNTAX_ERROR if the data in the message is invalid.

Called By:	process_command	
Routines Called:	get_thing printf	
Parameters:	INT	cmd
Returns:	rtn_val	

2.2.4.2 cig_2d_setup.c

The cig_2d_setup function is the 2-D overlay setup handler. This function is called by db_mcc_setup if the message from the Simulation Host is MSG_CIG_CTL - C_START_2D_SETUP, and a Force board is present.

The function call is **cig_2d_setup()**. cig_2d_setup does the following:

- Allocates memory for the setup.
- Starts the 2-D compiler by calling compile_2d.
- Deallocates the memory when the compiler is finished.

Called By:	db_mcc_setup
Routines Called:	calloc compile_2d free printf
Parameters:	none
Returns:	none

2.2.4.3 cig_comp_2d.c (compile_2d)

The cig_comp_2d.c CSU contains one function, compile_2d. This function is the main driver for the 2-D database compiler. compile_2d is responsible for processing the 2-D setup messages (MSG_2D_SETUP) sent from the Simulation Host. Each message represents one line in the ASCII 2-D database file.

The function call is **compile_2d()**. compile_2d does the following:

- Calls init_stuff to initialize various compiler variables.
- Calls get_msg_2d to get each line of the input file.
- Calls process_command to process each command from the input file.
- Checks for errors from process_command.
- Calls linkup to set up the window pointers and write the data to the GSP.
- Reports the number of errors detected during the compile.
- Cleans up and quits.

Called By:	cig_2d_setup
Routines Called:	get_msg_2d init_stuff linkup printf process_command
Parameters:	none
Returns:	none

2.2.4.4 cig_getm_2d.c (get_msg_2d)

The cig_getm_2d.c CSU contains one function, get_msg_2d. This function gets the next 2-D message from the input file and sets a pointer to it for compile_2d.

Each MSG_2D_SETUP message received from the Simulation Host represents one line of data in the ASCII input file. Each setup message is followed by a MSG_END message, making the MSG_2D_SETUP message the only message in the packet. get_msg_2d calls EXCHANGE_DATA to exchange packets each time a MSG_END message is detected. The full sequence is terminated by a MSG_CIG_CTL - C_STOP message.

The function call is **get_msg_2d()**.

The *msg_code* returned by the function is one of the following:

- CONTINUE_2D_SETUP if a valid 2-D setup message is found.
- STOP_2D_SETUP if a CIG Control-Stop message is detected.
- INVALID_2D_SETUP if an unknown message is detected.

Called By: compile_2d
get_thing

Routines Called: EXCHANGE_DATA
SYSERR

Parameters: none

Returns: msg_code

2.2.4.5 cig_link_2d.c (linkup)

The cig_link_2d.c CSU contains one function, linkup. This function is responsible for setting up window pointers and allocating available MPV (Micro Processor Video) memory for windows. It also downloads the data to GSP memory.

The function call is **linkup()**. linkup does the following:

- Calculates base addresses and table sizes for all information.
- Outputs the following information to stdout: component pointers table base address and size, window descriptor table base address and size, component descriptor table base address and size, allocatable window base address and maximum size, and base program address. See Figure 2-11 for a sample output.
- Sets up the screen window area (this should not vary).
- Changes the component pointers to absolute addresses.
- Allocates space for the dynamic polygon buffer areas.
- Sets the allocatable window area to the space following the component descriptor table.
- Allocates space for all windows and sets the window pointers.
- Downloads all data to GSP memory via the Force control register.

If the offline version of linkup is run, it writes all 2-D overlay data (header, component pointer table, window descriptor table, and component descriptor table) to the 2-D binary database file. The binary file can then be copied to the CIG and downloaded to GSP memory at a later time.

Figure 2-11 is a sample of the output generated by linkup.

```
file data2d_itl.0400    - Compiler output from:
compile_2d  data2d_ita.0400  data_2d_itb.0400 > data2d_itl.0400

BBN Systems and Technologies  Graphics Technology Division
2D Database Compiler  Date Thu Nov 17 15:23:31 PST 1988  Version: 0400
Link step starting ...
BASE COMPONENT POINTERS ADDRESS:                     0x07804100
  size of component pointer table:                   0x00000800
BASE WINDOW DESCRIPTOR TABLE ADDRESS:                0x07804900
  size of window descriptor table:                   0x00000800
BASE COMPONENT DESCRIP TABLE ADDRESS:                0x07805100
  size of component_descriptor_table:                0x000074d0
BASE ALLOCATABLE WINDOW ADDRESS:                     0x0780c5e0
  maximum size of allocatable area:                  0x00373a20
BASE PROGRAM ADDRESS:                                0x07b80000
   Allocating Dynamic Poly 0x3  at 0x780c5e0
   Next Available Address: 0x780ec20
  Space used: 0x2640  Space available: 0x3713e0
   Allocating Dynamic Poly 0x4  at 0x780ec20
   Next Available Address: 0x780ed40
  Space used: 0x2760  Space available: 0x3712c0
   Window 0x1 Allocated at GSP address: 0x780ed50
   Next Available Address: 0x78b6d50
  Space used: 0xaa760  Space available: 0x2c92b0
Compile finished -- Number of Errors = 0
```

Figure 2-11. Sample 2-D Compiler Output

Called By:	compile_2d
Routines Called:	DOWNLOAD_DATA printf TRIGGER_FORCE WAIT_FORCE
Parameters:	none
Returns:	none

2.2.4.6 comp.c (setup_comp_start)

The comp.c CSU contains one function, setup_comp_start. This function is responsible for placing component start data into the component descriptor table.

The function call is **setup_comp_start(cmd)**, where *cmd* is the command (N_COMP_START) passed by process_command.

setup_comp_start does the following:

- Calls get_thing to retrieve the component number, color, channel number, plane (foreground or background), window number, static/dynamic parameter, and rotation/translation values.
- Determines if the component descriptor table has room available.
- Places a pointer to this component in the component pointer table.
- Places all of the component data in the component descriptor table.

setup_comp_start provides some defaults if invalid parameters are encountered. The default color is white, the default plane is background, and the default static/dynamic parameter is static.

The *rtn_val* returned by the function is one of the following:

- SUCCESS if the data is added to the table successfully.
- COMPONENT_DESCRIPTOR_TBL_FULL if the table does not have enough room for the new data.
- INVALID_PARAMETERS if any of the component parameters provided is out of range.

Called By:	process_command	
Routines Called:	get_thing printf strcmp	
Parameters:	INT	cmd
Returns:	rtn_val	

2.2.4.7 draw_line.c (setup_draw_line)

The draw_line.c CSU contains one function, setup_draw_line. This function is responsible for updating line data in the component descriptor table.

The function call is **setup_draw_line(cmd)**, where *cmd* is the command (N_DRAW_LINE) passed by process_command.

setup_draw_line does the following:

- Calls get_thing to retrieve the parameters associated with this primitive.
- Determines if the component descriptor table has room available.
- Places the line's starting point x (column) and y (row), and the ending point x and y, into the component descriptor table.

The *rtn_val* returned by the function is one of the following:

- SUCCESS if the data is added to the table successfully.
- COMPONENT_DESCRIPTOR_TBL_FULL if the table does not have enough room for the new data.

- SYNTAX_ERROR if the data in the message is invalid.

Called By:	process_command
Routines Called:	get_thing printf
Parameters:	INT cmd
Returns:	rtn_val

2.2.4.8 get_thing.c

The get_thing function scans input lines for a specified number of data items of a specified type.

The function call is **get_thing(type, number)**, where:

type is the type of item (DATA_TYPE, COMMAND_TYPE, or TEXT_TYPE)
number is the number of items to be read

get_thing processes data as follows:

- Blank spaces and tab characters are discarded.
- If a digit is found and *type* is DATA_TYPE, get_thing sets a pointer to the data.
- If an alpha character is found and *type* is COMMAND_TYPE, get_thing sets a pointer to the command.
- If a quote character is found and *type* is TEXT_TYPE, get_thing sets a pointer to the text.
- If the end of line or comment is found, get_thing reads the next line.

This process continues until an error occurs or the specified number of items are read. The *rtn_val* returned by the function is one of the following:

- SUCCESS if the items were read successfully.
- SYNTAX_ERROR if unexpected data was found.

Called By:	process_command setup_bit_blt setup_comp_start setup_define_string setup_define_window setup_draw_line setup_oval_rectangle setup_poly setup_text

Routines Called:	get_msg_2d isalpha isdigit printf	
Parameters:	INT INT	type number
Returns:	rtn_val	

2.2.4.9 init_stuff.c

The init_stuff function initializes the following global data for the 2-D compilation process:

- Window descriptor table
- Allocation list
- Component pointer table
- Component descriptor table

The function call is **init_stuff()**.

Called By:	compile_2d
Routines Called:	none
Parameters:	none
Returns:	none

2.2.4.10 oval_rect.c (setup_oval_rectangle)

The oval_rect.c CSU contains one function, setup_oval_rectangle. This function is responsible for updating oval and rectangle data in the component descriptor table.

The function call is **setup_oval_rectangle (cmd)**, where *cmd* is the command (N_DRAW_OVAL, N_FILL_OVAL, N_DRAW_RECT, or N_FILL_RECT) passed by process_command.

setup_oval_rectangle does the following:

- Calls get_thing to retrieve the data in the message.
- Determines if the component descriptor table has room available.
- Places the object's width and height into the component descriptor table.
- Places the object's x (column of the upper left corner) and y (row of the upper left corner) coordinates into the component descriptor table.

The *rtn_val* returned by the function is one of the following:

- SUCCESS if the data is added to the table successfully.
- COMPONENT_DESCRIPTOR_TBL_FULL if the table does not have enough room for the new data.
- SYNTAX_ERROR if the data could not be processed.

Called By:	process_command	
Routines Called:	get_thing printf	
Parameters:	INT	cmd
Returns:	rtn_val	

2.2.4.11 poly.c (setup_poly)

The poly.c CSU contains one function, setup_poly. This function is responsible for updating polygon data in the component descriptor table.

The function call is **setup_poly(cmd)**, where *cmd* is the command (N_POLYLINE or N_FILL_POLY) passed by *process_command.*

setup_poly does the following:

- Calls get_thing to retrieve the data in the message.
- Determines if the component descriptor table has room available.
- Places the polygon's line and point data into the component descriptor table.

The *rtn_val* returned by the function is one of the following:

- SUCCESS if the data is added to the table successfully.
- COMPONENT_DESCRIPTOR_TBL_FULL if the table does not have enough room for the new data.
- SYNTAX_ERROR if the data in the message could not be processed.

Called By:	process_command	
Routines Called:	get_thing printf	
Parameters:	INT	cmd
Returns:	rtn_val	

2.2.4.12 proc_cmd.c (process_command)

The proc_cmd.c CSU contains one function, process_command. This function is responsible for retrieving a command string from get_thing, then calling the appropriate setup routine.

The function call is **process_command()**. process_command does the following:

- Calls get_thing to retrieve a command string.
- Compares the string with each possible command to determine which it is.
- When a match is found, calls the applicable setup routine.

The loop is repeated until all commands in the input file have been retrieved. The commands processed by process_command, and the setup function it calls for each, are listed below.

Command	Function Called(cmd)
A_BIT_BLT or B_BIT_BLT	setup_bit_blt(N_BIT_BLT)
A_COMP_START or B_COMP_START	setup_comp_start(N_COMP_START)
A_DEFINE_STRING or B_DEFINE_STRING	setup_define_string(N_DEFINE_STRING)
A_DEFINE_WINDOW or B_DEFINE_WINDOW	setup_define_window(N_DEFINE_WINDOW)
A_DRAW_LINE or B_DRAW_LINE	setup_draw_line(N_DRAW_LINE)
A_DRAW_OVAL or B_DRAW_OVAL	setup_oval_rectangle(N_DRAW_OVAL)
A_DRAW_RECT or B_DRAW_RECT	setup_oval_rectangle(N_DRAW_RECT)
A_ENDCOMP or B_ENDCOMP	none
A_FILL_OVAL or B_FILL_OVAL	setup_oval_rectangle(N_FILL_OVAL)
A_FILL_POLY or B_FILL_POLY	setup_poly(N_FILL_POLY)
A_FILL_RECT or B_FILL_RECT	setup_oval_rectangle(N_FILL_RECT)
A_POLYLINE or B_POLYLINE	setup_poly(N_POLYLINE)
A_TEXT or B_TEXT	setup_text(N_TEXT)

process_command keeps track of the number of errors returned by the setup functions. If the number of errors exceeds MAX_COMPILE_ERRORS (defined in defines_2d.h), process_command returns a *rtn_val* of TOO_MANY_ERRORS. This causes compile_2d to terminate the compile.

Called By: compile_2d

Routines Called: get_thing
printf
setup_bit_blt
setup_comp_start
setup_define_string

setup_define_window
setup_draw_line
setup_oval_rectangle
setup_poly
setup_text
strcmp

Parameters:	none
Returns:	rtn_val

2.2.4.13 string.c (setup_define_string)

The string.c CSU contains one function, setup_define_string. This function is responsible for placing initial string data into the component descriptor table.

The function call is **setup_define_string(cmd)**, where *cmd* is the command (N_DEFINE_STRING) passed by process_command.

setup_define_string does the following:

- Calls get_thing to retrieve the data from the message.
- Verifies that component start data has been entered into the component descriptor table.
- Determines whether the component descriptor table has room available.
- Places the string's font, x and y coordinates, and character data into the component descriptor table.

The *rtn_val* returned by the function is one of the following:

- SUCCESS if the data is added to the table successfully.
- COMPONENT_DESCRIPTOR_TBL_FULL if the table does not have enough room for the new data.
- SYNTAX_ERROR if if the string exceeds the maximum length allowed, the string contains a non-ASCII character, or the data in the message cannot be processed.

Called By:	process_command	
Routines Called:	get_thing printf strlen	
Parameters:	INT	cmd
Returns:	rtn_val	

2.2.4.14 text.c (setup_text)

The text.c CSU contains one function, setup_text. This function is responsible for placing fixed-length text data into the component descriptor table.

The function call is **setup_text(cmd)**, where *cmd* is the command (N_TEXT) passed by process_command. setup_text does the following:

- Calls get_thing to retrieve the data from the message.
- Verifies that the component descriptor table has room available.
- Places the text's font, x coordinate (lower left column), y coordinate (lower left row), and character string into the component descriptor table.

The *rtn_val* returned by the function is one of the following:

- SUCCESS if the data is added to the table successfully.
- COMPONENT_DESCRIPTOR_TBL_FULL if the table does not have enough room for the new data.
- SYNTAX_ERROR if a non-ASCII character is detected in the text string, or if the data in the message cannot be processed.

Called By:	process_command	
Routines Called:	get_thing printf strlen	
Parameters:	INT	cmd
Returns:	rtn_val	

2.2.4.15 window.c (setup_define_window)

The window.c CSU contains one function, setup_define_window. This function is responsible for placing window data into the window descriptor table.

The function call is **setup_define_window(cmd)**, where *cmd* is the command (N_DEFINE_WINDOW) passed by process_command.

setup_define_window does the following:

- Calls get_thing to retrieve the data from the message.
- Verifies that the parameters are valid.
- Computes the window's pitch and conversion factor. (See table below.)
- Places all window parameters (number of horizontal pixels, number of vertical pixels, pitch, and GSP conversion factor) into the window array structure.

- Places the window number into the allocation list so linkup can allocate memory for the window.

Pitch and conversion factors are computed as shown below.

# horizontal pixels (hex)	pitch (hex)	count (dec)	conversion factor (dec)
4001-8000	8000	15	16
2001-4000	4000	14	17
1001-2000	2000	13	18
801-1000	1000	12	19
401-800	800	11	20
201-400	400	10	21
101-200	200	9	22
80-100	100	8	23
41-80	80	7	24
21-40	40	6	25
11-20	20	5	26
8-10	10	4	27
4-8	8	3	28
2-4	4	2	29
1-2	2	1	30
1-1	1	0	31

The *rtn_val* returned by the function is one of the following:

- SUCCESS if the data is added to the table successfully.
- INVALID_WINDOW_NUMBER if the window number is out of range.
- INVALID_WINDOW_DX if the window's width is out of range.
- INVALID_WINDOW_DY if the window's height is out of range.
- WINDOW_PITCH_TOO_LARGE if the window's pitch is out of range.
- SYNTAX_ERROR if the data in the message cannot be processed.

Called By: process_command

Routines Called: get_thing
printf

Parameters: INT cmd

Returns: rtn_val

2.3 Database Management (ROWCOL_RD) CSC

The Database Management CSC is responsible for determining whether new rows and/or columns need to be read from the terrain database into active area memory for hardware, local terrain, and Ballistics use.

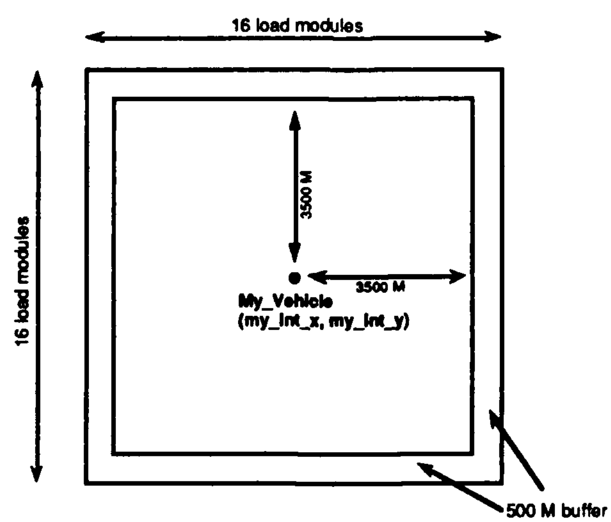
The terrain database, which is stored in the CIG, describes the entire terrain that can be displayed in the simulation. It also contains the graphic information used to display vehicles, houses, trees, hills, and other objects in the terrain.

The items stored in the terrain database are represented by connected polygons that are three-dimensional images. The polygons are grouped into compacted data structures such as terrain grids, polygon models, and stamp arrays. They are further grouped into unique static objects (rivers, roads, and other features that appear only once in the database) and generic models (houses, trees, vehicles, and other features that commonly recur in the database).

The terrain database is divided into units called load modules. One load module contains the instructions and data required to process a one-half kilometer square area of static objects. Each load module contains all the roads, rivers, terrains, buildings, and other features within a 500 by 500 meter area. The load modules in the terrain database are organized in rows and columns. The total size of the database is variable.

Each load module is divided into four areas called grids. Each grid is a 125M by 125M square.

Active area memory (AAM) contains the subset of the local terrain that can be viewed and interacted with at a given point in time by the simulation. The AAM stores an 8K by 8K area centered around the simulation vehicle. This provides a viewing range of 3500 meters in each direction, with a 500-meter buffer along each edge. The AAM contains 256 load modules (16 rows by 16 columns).



As the simulated vehicle moves toward an edge of active area memory, the Database Management CSC brings in new load modules from the terrain database, overwriting those areas that the vehicle is moving away from. The objective of this process is to keep the simulated vehicle in the center of active area memory.

Active area memory can be thought of as a window that moves over the terrain database. As the vehicle travels east, for example, the window must be moved east to keep the vehicle in the center. To do this, Database Management determines what column in the database lies east of the current east boundary of AAM. It then reads part of that column (the 16 load modules in the column that lie between AAM's north and south boundaries) into AAM. Finally, it shifts the west boundary of AAM over one column.

With very large terrain databases, load module blocking can be enabled. One load module block contains four load modules (two rows by two columns). Therefore, one load module block is 1000 meters by 1000 meters, or a one-kilometer square area. Load module blocking increases the amount of terrain that can be loaded into active area memory to 16K by 16K. It also doubles the viewing range of the simulated vehicle (from 3500 meters to 7000 meters).

Figure 2-12 identifies the CSUs in the Database Management CSC. The functions performed by these CSUs are described in this section.

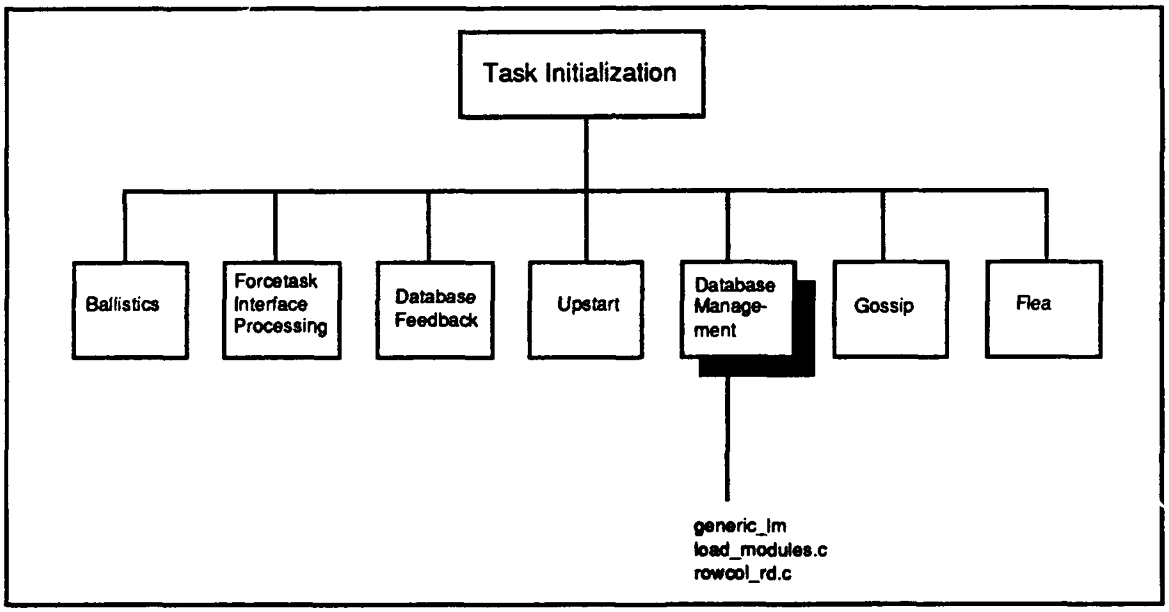


Figure 2-12. Database Management CSUs

2.3.1 generic_lm.c

The generic_lm.c CSU is used to initialize and generate a generic load module containing one ocean polygon. This allows a system to go beyond the defined database boundaries but still retain some orientation reference.

This CSU contains two functions:

- init_generic_lm
- generic_lm

2.3.1.1 init_generic_lm

The init_generic_lm function initializes a generic load module.

The function call is **init_generic_lm(view_range)**, where *view_range* is the viewing distance (3500 or 7000).

init_generic_lm does the following:

- Generates the load module header.
- Generates the required DTP commands.
- Generates the grid components.

Called By:	load_modules	
Routines Called:	none	
Parameters:	INT_4	view_range
Returns:	none	

2.3.1.2 generic_lm

The generic_lm function generates the generic load module. It copies the load module to memory, then updates the load module header, DTP, and grid components.

The function call is **generic_lm(swx, swy, centoff, memptr)**, where:

swx is the x coordinate of the load module's southwest corner
swy is the y coordinate of the load module's southwest corner
centoff is the offset to the center of the load module
memptr is a pointer to the AAM location for the new load module

Called By:	getside	
Routines Called:	none	
Parameters:	INT_4	swx
	INT_4	swy
	REAL_4	centoff
	GENLM	*memptr

Returns: none

2.3.2 load_modules.c

The functions in load_modules.c are used to load new active area rows and columns from the terrain database when required. These functions are:

- getlmdp
- getside
- whatdirptr
- load_modules

2.3.2.1 getlmdp

The getlmdp function gets a load module's disk pointer from the database.

The function call is **getlmdp(xmod, ymod, rowcol_dbfd)**, where:

xmod is the load module array number x
ymod is the load module array number y
rowcol_dbfd is the file descriptor for the terrain database

The function returns the disk pointer if successful, or 0 if the load module is not in the database. If 0 is returned, getside calls generic_lm to get a generic load module.

Called By:	getside	
Routines Called:	XLSEEK XREAD	
Parameters:	INT_4 INT_4 int	xmod ymod rowcol_dbfd
Returns:	dbde.lm_loc	

2.3.2.2 getside

The getside function loads part of a row or column from the terrain database into active area memory. The number of load modules in the row or column that are actually loaded into AAM is 16 (the normal height/width of AAM) or 32 (the height/width of AAM if load module blocking is enabled).

The function call is **getside(lmdloc, xmod, ymod, xinc, yinc, diroff, zeroit, rowcol_dbfd)**, where:

lmdloc is a pointer to the first load module on disk
xmod is the first load module's array number x (west column)
ymod is the first load module's array number y (south row)
xinc is the load module's array number increment x
yinc is the load module's array number increment y
diroff is a byte offset to the directory pointer in the load module header
zeroit is a flag used to determine when the running average load module centroid should be zeroed
rowcol_dbfd is the file descriptor for the terrain database

getside does the following for each new load module it loads into AAM:

- Sets the load module number and gets its AAM address.
- Checks that the load module is in the database and gets its disk pointer.
- If the load module is not in the database, calls generic_lm to get a generic load module.
- Informs Ballistics of the new load module (by pushing a MSG_B0_LM_READ message onto the Ballistics message queue).
- Updates the field-of-view tables for the new load module.

Called By:	load_modules rowcol_rd	
Routines Called:	getlmdp generic_lm mx_push XLSEEK XREAD	
Parameters:	WORD	lmdloc
	INT_4	xmod
	INT_4	ymod
	INT_4	xinc
	INT_4	yinc
	WORD	diroff
	WORD	zeroit
	int	rowcol_dbfd
Returns:	none	

2.3.2.3 whatdirptr

The whatdirptr function finds the direction pointer for the load module at a specified location in a specified direction.

The function call is **whatdirptr(xmod, ymod, diroff)**, where:

xmod is the load module's array number x (west column)
ymod is the load module's array number y (south row)

diroff is the byte offset to the direction pointer in the load module header

Called By:	load_modules rowcol_rd	
Routines Called:	none	
Parameters:	INT_4 INT_4 WORD	xmod ymod diroff
Returns:	<direction pointer>	

2.3.2.4 load_modules

The load_modules function loads a portion of the terrain database into AAM. load_modules is called when AAM needs to be completed loaded. It is called by load_dbase to load the initial load modules into active area memory. During a simulation, load_modules is called by rowcol_rd if the simulated vehicle is detected to be out of viewing range of active area memory. Specifically, the vehicle must be more than one-half the width of AAM outside its boundaries. In this instance, none of the terrain that is currently visible to the vehicle is in AAM — usually, this is due to "warping" across the terrain. rowcol_rd then calls load_modules to rebuild all of AAM based on the vehicle's current location.

The function call is **load_modules(file_descriptor)**, where *file_descriptor* identifies the database file to be read.

load_modules does the following:

- Initializes direction offsets.
- Calls init_generic_lm to initialize a generic load module for the applicable viewing range.
- Calculates the southwest corner of AAM based on the current coordinates of the simulated vehicle.
- Calculates the four borders of AAM.
- Reads each AAM row (south to north) from west to east, calling getside to load the appropriate load modules from the database.
- Calls whatdirptr to find the direction pointer after the first row of load modules is loaded.
- After reaching the northernmost row, resets the address of the south border.

Called By:	load_dbase rowcol_rd
Routines Called:	getside init_generic_lm

whatdirptr

Parameters:	INT	file_descriptor
Returns:	none	

2.3.3 rowcol_rd.c

The rowcol_rd.c CSU contains two functions:

- main (for Butterfly compatibility only)
- rowcol_rd

2.3.3.1 main

The main function invokes the rowcol_rd function. It requires one argument: *bvme_id*, which identifies the Butterfly-VME interface. This function is required for Butterfly compatibility only.

Called By:	none	
Routines Called:	atoi Find_Value Make_Event map_vme Name_Bind printf remap_vme	
Parameters:	int char	argc *argv[]
Returns:	none	

2.3.3.2 rowcol_rd

The rowcol_rd function determines whether a new row or column of the database needs to be read into active area memory. This task is started automatically by rtt during the task initialization state.

rowcol_rd waits until simulation posts a message to its mailbox, indicating that database management is required. It then does the following:

- Initializes direction offsets.

- Tells Ballistics where the southwest corner of active area memory is, by pushing the MSG_B0_AAM_SW_CORNER message onto the Ballistics message queue.
- Checks to see if the simulated vehicle is out of viewing range of AAM (i.e., is beyond an AAM boundary by a distance of more than one-half AAM width). If so, calls load_modules to reload all of AAM from the terrain database.
- Checks to see if the simulated vehicle is inside AAM, or outside but within viewing range of it. If so, compares the coordinates of the vehicle's centroid to the center of AAM.
- If the vehicle is detected to be off-center, calls whatdirptr and getside to load a new row or column in the needed direction. For example, if the vehicle is detected to be too far away from the west boundary (i.e., is east of AAM center), a column is added to the east side and deleted from the west. This has the effect of shifting all of AAM east by one column.
- Updates the necessary database data variables to reflect the change to AAM boundaries.
- Checks to make sure all static vehicles are within the active area.

Called By:	none
Routines Called:	getside load_modules mx_push sc_pend sc_post whatdirptr XCLOSE
Parameters:	none
Returns:	none

2.4 Database Feedback (LOCAL_TERRAIN) CSC

The Database Feedback CSC builds new local terrain messages. These messages are used by the Simulation Host to provide collision detection with objects in the simulated environment, and to calculate the dynamics of the vehicle in operation.

A local terrain message contains data describing the terrain, roads, rivers, and buildings that lie within the four grids surrounding the simulated vehicle. (One grid is usually 125 meters per side. One load module is defined as four grids — two rows by two columns.)

A new local terrain message is sent to the Simulation Host every 32 frames. Each message contains the following:

- A header that specifies the number of polygon definitions and the number of bounding volumes (bvols) contained in the message.
- Polygons that describe the local terrain and the objects in it. These polygons are planar, convex, and three- or four-sided. Each polygon entry in the message specifies the soil type, priority code, minimum and maximum coordinates, and all polygon vertices in counter-clockwise order.
- Bounding volumes. A bvol definition contains one or more four-sided bounding boxes each of which has a planar, convex, polygonal base and a height (expressed in units on the z axis) for the volume given. Each bvol entry in the message specifies the bvol's height above the polygonal base, the bvol type identifier, the minimum and maximum coordinates, and the vertex list.

Local terrain messages can also be sent on demand from the Simulation Host, in response to a MSG_RTN_LT (return local terrain) message. This message is to be used by the MCC station only.

The CSUs in the Database Feedback CSC are identified in Figure 2-13. The functions performed by these CSUs are described in this section.

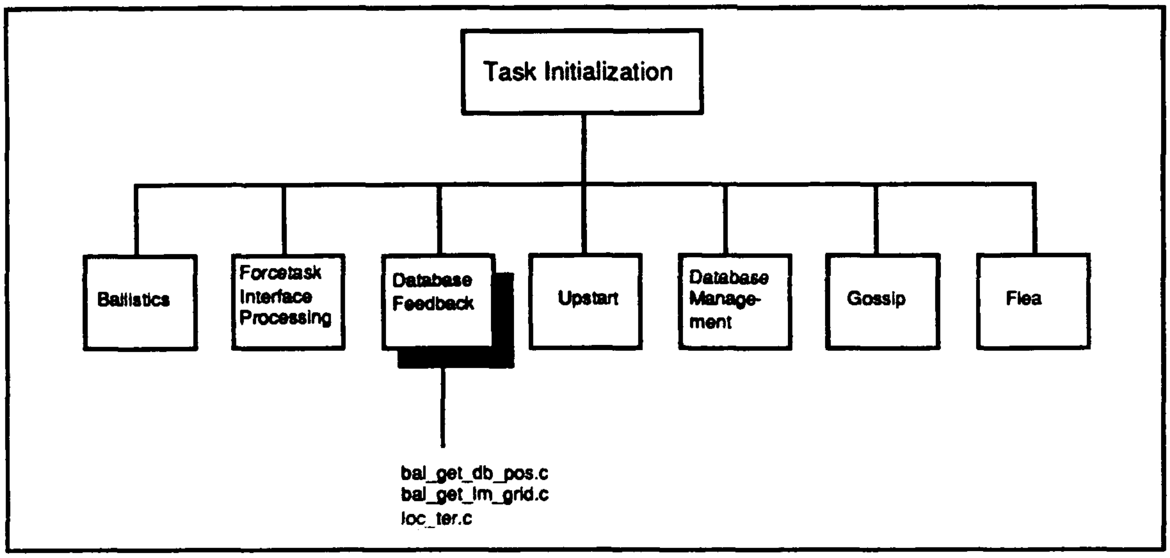


Figure 2-13. Database Feedback CSU

2.4.1 bal_get_db_pos.c

The bal_get_db_pos function finds the load module number and grid number of a given chord point. This function is called by local_terrain to determine the load module and grid of the simulated vehicle's current position.

The function call is **bal_get_db_pos(pcrd, lm_width, lm_per_side)**, where:

pcrd is a pointer to the chord data
lm_width is the width of a load module
lm_per_side is the number of load modules in a row or column of AAM

bal_get_db_pos calls FIND_LM to determine the load module for the x and y coordinates provided by local_terrain (in the chord data). It then calculates which grid the vehicle occupies within the load module. The load module and grid number are placed in the chord data structure.

Called By:	local_terrain	
Routines Called:	FIND_LM	
Parameters:	CHORD_DATA INT_4 INT_4	*pcrd lm_width lm_per_side
Returns:	none	

2.4.2 bal_get_lm_grid.c

The bal_get_lm_grid function finds the load modules and grids in the database that are intersected by a chord. This function is called by local_terrain to determine what four grids lie around the simulated vehicle. (One grid is 125 meters wide.)

The function call is **bal_get_lm_grid(pcrd, lm_per_side, lm_size, lm_base_addr, bal_search, dvl_search, lm_width)**, where:

pcrd is a pointer to the chord data
lm_per_side is the number of load modules in a row or column of AAM
lm_size is the size in bytes of a load module
lm_base_addr is the load module's base address
bal_search is the array in which to store load module offsets and grid words
dvl_search is the array in which to store dynamic module path data
lm_width is the width of a load module

The function returns 1 if it is successful, or 0 if an illegal chord (one longer than 125 meters) is detected.

Called By:	local_terrain	
Routines Called:	none	
Parameters:	CHORD_DATA	pcrd[]
	INT_4	lm_per_side
	INT_4	lm_size
	INT_4	lm_base_addr
	SEARCH_LIST	bal_search[]
	INT_4	dvl_search[]
	INT_4	lm_width
Returns:	1 (TRUE)	
	0 (FALSE)	

2.4.3 loc_ter.c

The loc_ter.c CSU contains two functions:

- main (for Butterfly compatibility only)
- local_terrain

2.4.3.1 main

The main function invokes the local_terrain function. It requires one argument: *bvme_id*, which identifies the Butterfly-VME interface. This function is required for Butterfly compatibility.

Called By:	none	
Routines Called:	atoi Find_Value local_terrain Make_Event map_vme Name_Bind printf remap_vme	
Parameters:	int char	argc *argv[]
Returns:	-1	

2.4.3.2 local_terrain

The local_terrain function builds a local terrain message, based on the simulated vehicle's current position, for transmission to the Simulation Host. The local_terrain task is loaded by rtt during the task initialization state. It is suspended until simulation posts a message to its mailbox (LOCAL_TERRAIN_MB).

The first frame at which a local terrain message is created, and the interval at which new messages are generated, are defined in the memory_map_defines.h include file. Currently, the first frame is set to 16 and the interval is set to 32.

The simulation vehicle's current position (*my_int_x*, *my_int_y*) is stored in the viewport positions array, which is maintained by process_vppos. local_terrain assumes that the vehicle's coordinates have just been updated.

When woken up by simulation, local_terrain does the following:

- Initializes the local terrain output buffer header (version and level).
- Calls read_watch to get the timer tick count.
- Calls bal_get_db_pos to find the simulated vehicle's current load module number and grid number.
- Calls bal_get_lm_grid to find the four grids that surround the simulated vehicle.
- Determines whether a new local terrain message needs to be built (i.e., if the simulated vehicle's position has changed since the last local terrain message).
- If the vehicle has moved, reinitializes the local terrain output buffer.

- Searches the four grids that lie around the simulated vehicle for polygons, and builds the polygon portion of the message.
- Searches the four grids that lie around the simulated vehicle for polygon components, and builds the polygon component portion of the message.
- Searches the four grids that lie around the simulated vehicle for bounding volumes, and builds the bvol portion of the message.
- Sets a pointer to the new local terrain message data.
- Creates the message header: message size, message type (MSG_LOCAL_TERRAIN, and the total number of polygons and bvols in the message.
- Posts a message to the RTN_TERRAIN_MB mailbox.

Called By:	none
Routines Called:	bal_get_db_pos bal_get_lm_grid FXTO881 FXTOFL read_watch sc_pend sc_post
Parameters:	none
Returns:	none

2.5 Ballistics Processing (BALLISTICS) CSC

The Ballistics Processing CSC is responsible for the following:

- Detecting intersections with the terrain database and the currently viewable models (static and dynamic vehicles).
- Processing round data and returning hit or miss information to the real-time software.
- Processing trajectory chord data and returning hit or miss information to the real-time software.

The following points apply to intersection calculations:

- When determining whether a given trajectory intersects with a model or the terrain, Ballistics treats the trajectory as a series of consecutive chords. Each chord is a maximum of 115 meters. All computations are performed on the chords.
- Intersections with models are calculated with the bounding volume surrounding the model or its articulated part, not with the model itself. A bounding volume, or bvol, is the volume of the bounding box that is used to enclose a model in the simulation environment. The use of bvols reduces the number of surfaces that Ballistics must deal with. An intersection with any surface of any bvol belonging to a model is considered an intersection with that model.
- Intersections with the terrain are calculated with polygons that have the local terrain flag and/or the Ballistics flag set true.

Ballistics is loaded and started by upstart, then put into the run state by simulation. The communication between the real-time software and Ballistics consists of the following:

- Messages sent from the Simulation Host. For example, a message may tell Ballistics that a round has been fired, or that a static vehicle has been added to the local terrain. Each Ballistics message is received by simulation, which pushes it onto the Ballistics message queue. Ballistics processes the message (which typically involves computing whether any model or terrain in the database was hit), then returns a hit or miss message if applicable. Messages returned from Ballistics are removed from the message queue by simulation, which sends them to the Simulation Host.
- Once per frame, simulation notifies Ballistics that a frame interrupt has taken place, and informs it (via a MSG_B0_NEW_FRAME message) of the current frame count and the new status of all dynamic vehicles.
- When the getside task (called by load_modules) loads a new load module from disk into active area memory, it informs Ballistics using a MSG_B0_LM_READ message.

Ballistics Processing may be run on a master board or a slave board in the CIG, as follows:

Master
If the CIG has only one MVME133 board, it is the master that is used to run all of the real-time software, including Ballistics.

Slave
If the CIG has two MVME133 boards, the left board is the master that runs the real-time software. The right board is the slave that runs Ballistics. This configuration is used for high rate-of-fire weapons.

A CIG that interfaces to a Butterfly Simulation Host has only one MVME133 board, which is used to run Ballistics. The real-time software runs on the Butterfly itself.

Note: *The Dart Ballistics Processing board is no longer supported. References in the code to the Dart implementation can be disregarded.*

The Ballistics software that runs on a master board is very similar to the software that runs on a slave board. Most of the variations are identified in the code by the SLAVE133 compiler flag. The real-time software determines what type of Ballistics board is in the CIG, then loads the appropriate version of the Ballistics task.

The major data structures used in Ballistics Processing are the following:

Trajectory table directory
Contains one entry for each trajectory table. A trajectory table, which describes the trajectory for a specific type of round, consists of the trajectory type, frame rate, effect type, table size, and a pointer to the table's entries. Each trajectory table entry contains the trajectory's boresight x and y coordinates (with respect to the gun barrel).

Trajectory tables are predefined for certain round types. The Simulation Host may define trajectory tables for other round types.

Terrain model directory
Describes the models that are placed on the terrain (houses, telephone poles, water towers, etc.). Each entry defines the model type, bvol flag, component count, bvol count, model directory type, model radius, and the primary, secondary, and tertiary bvol indices.

Note: *The terrain model directory is not currently used. It is set up to accommodate future enhancements to the database.*

Terrain bvol directory
Describes the bounding volume for each terrain model. Each entry defines the model directory type, type id, the bvol's height above the poly-defining perimeter, and the perimeter defining the bvol polygon (its vertices).

Note: *The terrain bvol directory is not currently used. It is set up to accommodate future enhancements to the database.*

DED model directory
Describes the models in the dynamic elements database. Each entry defines the model type, bvol flag, component count, bvol count, model directory type, model radius, and the primary, secondary, and tertiary bvol indices.

DED bvol directory

Describes the bounding volume for each DED model. Each entry defines the bvol index, the model directory type, type id, the bvol's height above the poly-defining perimeter, and the perimeter defining the bvol polygon (its vertices).

Load module directory

Contains one entry for each load module in active area memory. Each load module entry contains the load module's cache flag, frame stamp, polygon count, maximum polygon height above the poly-defining perimeter, bvol count, and maximum bvol height above the poly-defining perimeter. Each load module entry also contains pointers to the polygon and bvol lists attached to that load module.

Static vehicle directory

Contains one entry for every load module in active area memory. Each entry points to a list of the static vehicles in that load module. Each entry in the static vehicle list contains the static vehicle's vehicle id, AAM partition index, component count, unique type, load module number, application-specific data (ASID), transformation matrix, rotation angles for the second component, and back and forward pointers.

Static vehicle entries that are not currently assigned to a load module are contained in the static vehicle free list. When the Simulation Host requests the addition of a static vehicle, Ballistics removes one from the free list and adds it to the proper load module list. When the Simulation Host specifies deletion of a static vehicle, Ballistics removes it from the load module and returns it to the free list. The free list is a mechanism for ensuring that the maximum number of static vehicles is not exceeded.

Polygon lists

Contain one entry for each polygon in a given load module in active area memory. Each entry contains the polygon's soil type, vertex count, priority, shade, minimum and maximum values, Ballistics flag, local terrain flag, grid location, and vertex list. Each load module in active area memory has its own polygon list.

Polygon entries that are not currently assigned to a load module are contained in the free polygon list. When a new load module is added to active area memory, Ballistics removes the required number of polygons from the free list and adds them to the new load module's polygon list. If the free list does not contain enough polygons for a new load module, Ballistics swaps out the least-recently-used load module. When a load module is removed from active area memory, Ballistics returns its polygons to the free list.

Bvol lists

Contain one entry for each bounding volume in a given load module in active area memory. Each entry contains the bvol's type id, distance above the poly-defining perimeter, vertex list, and grid location. Each load module in active area memory has its own bvol list.

bvol entries that are not currently assigned to a load module are contained in the free bvol list. When a new load module is added to active area memory, Ballistics removes the required number of bvols from the free list and adds them to the new load module's bvol list. If the free list does not contain enough bvols for a new load module, Ballistics swaps out the least-recently-used load module. When a load module is removed from active area memory, Ballistics returns its bvols to the free list.

Round list
Contains one entry for each active round. Each entry contains the round's active frame count, frame count, frame interval, trajectory entry index, trajectory table size, offset, trajectory pointer, points, and back and forward pointers.

Round entries that are not currently active are contained in the free round list. When the Simulation Host requests a new round, Ballistics removes one from the free list and adds it to the active list. After processing the round, Ballistics removes it from the active list and returns it to the free list. The free list is a mechanism for ensuring that the maximum number of rounds is not exceeded.

Ballistics Processing is divided into the following functional areas:

Ballistics Mainline
Initializes all Ballistics structures at start-up, and drives all Ballistics processing.

Ballistics Interface Message Processing
Processes the Ballistics messages received from the Simulation Host.

Ballistics Intersection Calculations
Calculates chord intersections to determine if anything in the simulated environment was hit by a round or trajectory. Acquires polygon and bounding volume information from the terrain database, and maintains the data in a cache using an LRU swapping algorithm. Also maintains static vehicles using a set of free lists.

Ballistics Message Queue Processing
Maintains the message queues used as the interface between Ballistics and the real-time software.

Figure 2-14 identifies the CSUs in the Ballistics CSC. The CSUs in each functional area are described in the following subsections, in the order listed above.

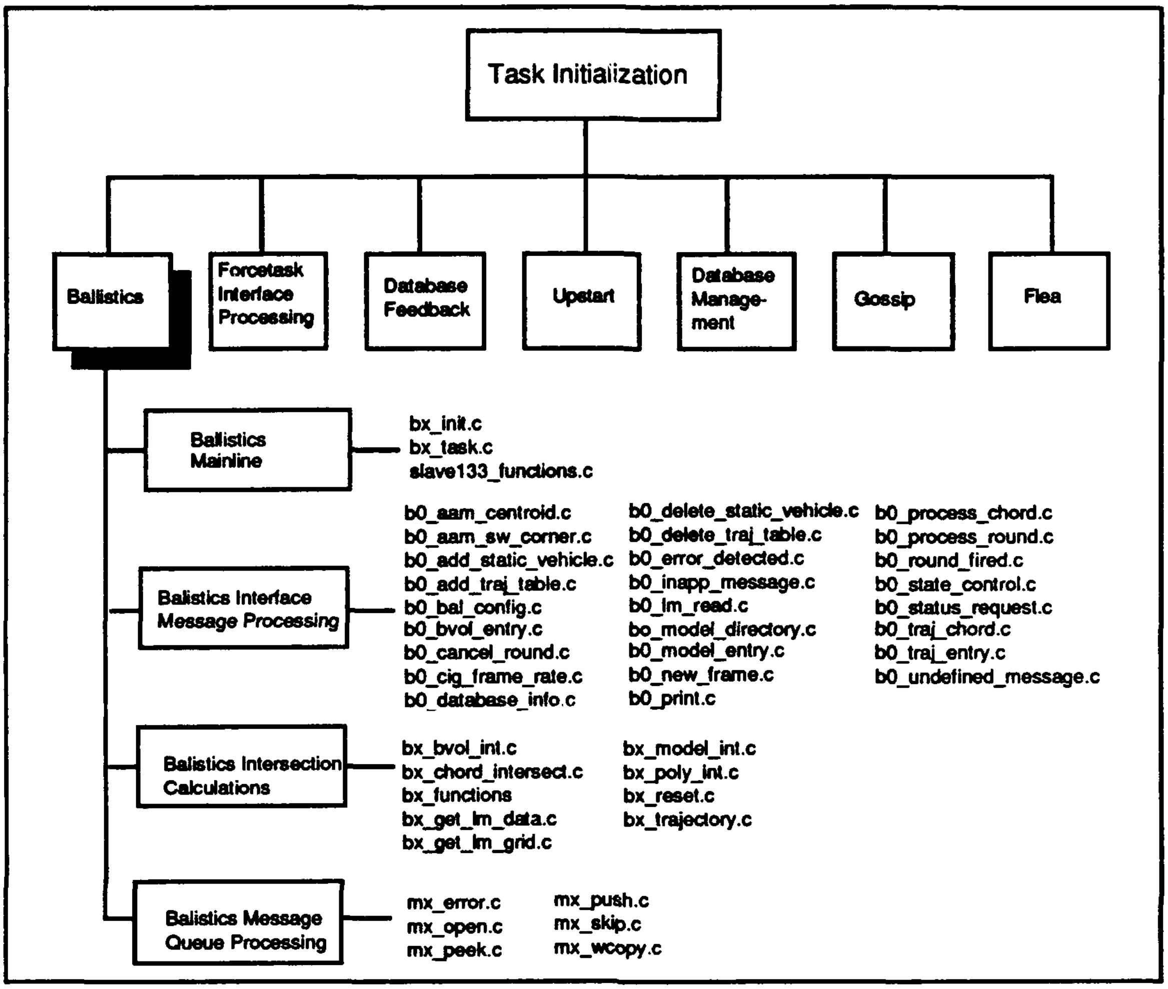


Figure 2-14. Ballistics Processing CSUs

2.5.1 Ballistics Mainline

This section describes the Ballistics Mainline component of the Ballistics Processing CSC. The CSUs in this component provide the functions that initialize and drive Ballistics Processing on the CIG.

2.5.1.1 bx147_main.c (main)

The main function in bx147_main.c is not used on the 120TX/T CIG. Information provided on this function in earlier releases of this document should be disregarded.

2.5.1.2 bx_init.c

The bx_init function is called by bx_task to initialize Ballistics. bx_init defines the message arrays (*G_init_message[]* and *G_run_message[]*) used by bx_task to process Ballistics messages. It also initializes the following structures:

- Terrain and dynamic elements database (DED) model directories.
- Terrain and DED bounding volume directories.
- Static vehicle list.
- Bounding volume cache list.
- Polygon cache list.
- Round list.
- Trajectory table directory and tables.
- Various pointers, lists, and temporary variables.

The function call is **bx_init()**.

Called By:	bx_task
Routines Called:	none
Parameters:	none
Returns:	none

2.5.1.3 bx_task.c

The bx_task function is the main Ballistics task. It is loaded into the task table by rtt during task initialization, and put into the run state by simulation.

bx_task does the following:

- Calls bx_init to initialize structures used by Ballistics.

- Locates the message queues used to communicate with the real-time software, and installs and opens them.
- Notifies the real-time software that Ballistics has started (via a MSG_B1_STATUS_RETURN message).
- Gives the real-time software the addresses of Ballistics global variables (via a MSG_B1_GLOBAL_ADDR message).
- Reads each Ballistics message in turn from the message queue.

Messages are pushed onto the Ballistics message queue by simulation. bx_task manages the message queue using the Ballistics Message Queue Processing functions (see section 2.5.4). When it pops a message from the stack, it calls the appropriate Ballistics Interface Message Processing routine (see section 2.5.2) to process it.

Called By: none

Routines Called:
b0_aam_sw_corner
b0_add_static_vehicle
b0_add_traj_table
b0_bal_config
b0_bvol_entry
b0_cancel_round
b0_cig_frame_rate
b0_database_info
b0_delete_static_vehicle
b0_delete_traj_table
b0_error_detected
b0_inapp_message
b0_lm_read
b0_model_directory
b0_model_entry
b0_new_frame
b0_print
b0_process_chord
b0_process_round
b0_round_fired
b0_state_control
b0_status_request
b0_traj_chord
b0_traj_entry
b0_undefined_message
bx_init
mx_error
mx_open
mx_peek
mx_push
mx_skip
printf
puts

Parameters: none

Returns:	none

2.5.1.4 slave133_functions.c

The slave133_functions.c CSU contains functions that are required to run Ballistics on a slave board. The functions contained in this CSU are the following:

- slave133_malloc
- free133

2.5.1.4.1 slave133_malloc

The slave133_malloc function allocates memory on the slave board. The MALLOC macro invokes slave133_malloc (instead of malloc) if Ballistics is running on a slave board.

The function call is **slave133_malloc(byte_count)**, where *byte_count* is the amount of memory to be allocated. The function returns a pointer to the beginning of the free area of memory as *head_P*.

Called By:	MALLOC	
Routines Called:	none	
Parameters:	WORD	byte_count
Returns:	head_P	

2.5.1.4.2 free133

The free133 function returns all memory allocated with slave133_malloc to the slave board's memory pool. This function is called by bx_reset to reclaim dynamic memory.

The function call is **free133()**.

Called By:	bx_reset
Routines Called:	none
Parameters:	none
Returns:	none

2.5.2 Ballistics Interface Message Processing

This section describes Ballistics Interface Message Processing, a major functional component of the Ballistics Processing CSC. It contains the functions that process the Ballistics messages that are received by the bx_task from the real-time software.

The Ballistics Interface Message Processing functions are defined as elements of arrays in bx_init. Two arrays are used: *G_init_message[]* and *G_run_message[]*. The messages in *G_init_message* are used to initialize Ballistics (e.g., define model entries or the trajectory table). The messages in *G_run_message* are used to respond to runtime messages (e.g., process rounds or manage static vehicles). The index into either array is the message code (*G_m_code*).

The complete processing mechanism is as follows:

1. The Simulation Host sends a Ballistics message.
2. simulation calls mx_push to push the message onto the Ballistics message queue. simulation sets the *message_code* to M_B0_<message>.
3. bx_task pops the message from the message queue.
4. bx_task indexes into *G_init_message[]* or *G_run_message[]* with the message code (*G_m_code*). It also passes a pointer to the message (*message_P*).
5. The function corresponding to the specified element in the specified array is called with the *message_P* parameter.

This method of invoking the Ballistics Interface Message Processing functions provides for faster processing than direct function calls.

Note that some of the messages sent from simulation to Ballistics do not originate from the Simulation Host. For example, simulation generates messages to start or stop Ballistics, and to tell Ballistics where active area memory is. The processing mechanism for such messages is the same as for those received from the Simulation Host.

Some Ballistics messages cause a return message. For example, a ROUND_FIRED message results in a HIT_RETURN or MISS message. The Ballistics Interface Message Processing function generates the response message and calls mx_push to push it onto the message queue with the *message_code* set to M_B1_<message>. simulation retrieves the message from the queue and processes it accordingly.

2.5.2.1 b0_aam_centroid.c

The b0_aam_centroid function is a stub for future expansion; it is not currently used.

The function call is **b0_aam_centroid()**. The function always returns 0.

2.5.2.2 b0_aam_sw_corner.c

The b0_aam_sw_corner function processes the message MSG_B0_AAM_SW_CORNER. This message is sent by simulation when Ballistics is first put into the run state. It is also sent by rowcol_rd whenever active area memory is relocated. The message gives Ballistics the coordinates of the southwest corner of active area memory. The b0_aam_sw_corner function calculates the coordinates of the northeast corner by adding twice the viewing range in each direction.

The function call is **b0_aam_sw_corner(message_P)**, where *message_P* is a pointer to the MSG_B0_AAM_SW_CORNER message.

The function always returns 0.

Called By:	bx_task	
Routines Called:	none	
Parameters:	MSG_B0_AAM_SW_CORNER	*message_P
Returns:	0	

2.5.2.3 b0_add_static_vehicle.c

The b0_add_static_vehicle function processes the MSG_B0_ADD_STATIC_VEHICLE message. This message is sent by simulation when the Simulation Host sends a message to add a new static vehicle to the local terrain. The message specifies the vehicle id, type, orientation, and position.

The function call is **b0_add_static_vehicle(message_P)**, where *message_P* is a pointer to the MSG_B0_ADD_STATIC_VEHICLE message.

The function returns a 0 if successful. It returns 1 if the vehicle's load module is out of range, the maximum vehicle limit has been reached, or the number of components (values used to determine the vehicle's orientation and position) is not 1 or 3.

Called By:	bx_task	
Routines Called:	BCOPY NEW_STAT_VEH	
Parameters:	MSG_B0_ADD_STATIC_VEHICLE	*message_P

Returns: 1
0

2.5.2.4 b0_add_traj_table.c

The b0_add_traj_table function processes the message MSG_B0_ADD_TRAJ_TABLE. This message is sent by db_mcc_setup when processing a MSG_TRAJ_TABLE_XFER message from the Simulation Host. This message is used to add trajectory tables. The message specifies the table's trajectory type, frame rate, effect type, and number of entries. Entries are added using the b0_traj_entry function.

The function call is **b0_add_traj_table(message_P)**, where *message_P* is a pointer to the MSG_B0_ADD_TRAJ_TABLE message.

The function returns 0 if successful, or -1 if the trajectory type is invalid.

Called By: bx_task

Routines Called: free
MALLOC

Parameters: MSG_B0_ADD_TRAJ_TABLE *message_P

Returns: -1
0

2.5.2.5 b0_bal_config.c

The b0_bal_config function processes the message MSG_B0_BAL_CONFIG. This message is sent by open_dbase to give Ballistics its initialized configuration parameters.

The function call is **b0_bal_config(message_P)**, where *message_P* is a pointer to the MSG_B0_BAL_CONFIG message.

The function always returns 0.

Called By: bx_task

Routines Called: BCOPY

Parameters: MSG_B0_BAL_CONFIG *message_P

Returns: 0

2.5.2.6 b0_bvol_entry.c

The b0_bvol_entry function processes the message MSG_B0_BVOL_ENTRY. This message is sent by download_bvols to to add bounding volumes to the terrain or DED model directory.

The function call is **b0_bvol_entry(message_P)**, where *message_P* is a pointer to the MSG_B0_BVOL_ENTRY message.

The function always returns 0.

Called By:	bx_task	
Routines Called:	BCOPY	
Parameters:	MSG_B0_BVOL_ENTRY	*message_P
Returns:	0	

2.5.2.7 b0_cancel_round.c

The b0_cancel_round function is a stub for future expansion; it is not currently implemented.

The function call is **b0_cancel_round()**. The function always returns 0.

2.5.2.8 b0_cig_frame_rate.c

The b0_cig_frame_rate function processes the message MSG_B0_CIG_FRAME_RATE. simulation sends this message to tell Ballistics the frame rate (15 or 30 Hz).

The function call is **b0_cig_frame_rate(message_P)**, where *message_P* is a pointer to the MSG_B0_CIG_FRAME_RATE message.

The function always returns 0.

Called By:	bx_task	
Routines Called:	none	
Parameters:	MSG_B0_CIG_FRAME_RATE	*message_P

Returns: 0

2.5.2.9 b0_database_info.c

The b0_database_info function processes the message MSG_B0_DATABASE_INFO. open_dbase sends this message after it initializes AAM partition information.

The function call is **b0_database_info (message_P)**, where *message_P* is a pointer to the MSG_B0_DATABASE_INFO message.

b0_database_info does the following:

- Allocates space for the load module tables.
- Loads the load module cache data.
- Sets up the table of load module addresses.

The function always returns 0.

Called By: bx_task

Routines Called: MALLOC

Parameters: MSG_B0_DATABASE_INFO *message_P

Returns: 0

2.5.2.10 b0_delete_static_vehicle.c

The b0_delete_static_vehicle function processes the message MSG_B0_DELETE_-STATIC_VEHICLE. simulation sends this message when it receives a MSG_STATICVEH_REM message from the Simulation Host. The message contains the vehicle id, type, and current position (x, y, and z coordinates) of the vehicle to be deleted from active area memory.

The function call is **b0_delete_static_vehicle(message_P)**, where *message_P* is a pointer to the MSG_B0_DELETE_STATIC_VEHICLE message.

The function returns 0 if the static vehicle is successfully deleted. It returns 1 if the specified vehicle not found in active area memory.

Called By: bx_task

Routines Called: DELETE_STAT_VEH
outhexl (if running on a slave board)
puts (if running on a slave board)

Parameters:	MSG_B0_DELETE_STATIC_VEHICLE	*message_P
Returns:	1 0	

2.5.2.11 b0_delete_traj_table.c

The b0_delete_traj_table function a stub for future enhancement; it is not currently implemented.

The function call is **b0_delete_traj_table()**. The function always returns 0.

2.5.2.12 b0_dummy.c

The b0_dummy function is a template for adding other b0_* functions; it is not called by any other function.

The function call is **b0_dummy()**. The function always returns 0.

2.5.2.13 b0_error_detected.c

The b0_error_detected function is a stub for future enhancement; it is not currently implemented.

The function call is **b0_error_detected()**. The function always returns 0.

2.5.2.14 b0_inapp_message.c

The b0_inapp_message function outputs the "*** Inappropriate Message ***" error for slave boards.

The function call is **b0_inapp_message()**. The function always returns 0.

Called By:	bx_task
Routines Called:	puts
Parameters:	none
Returns:	0

2.5.2.15 b0_lm_read.c

The b0_lm_read function processes the message MSG_B0_LM_READ for Ballistics. This message is sent by getside (in load_modules) to inform Ballistics of a new load module added to the local terrain.

The function call is **b0_lm_read(message_P)**, where *message_P* is a pointer to the MSG_B0_LM_READ message. The function always returns 0.

Called By:	bx_task	
Routines Called:	FREE_LM_CACHE	
Parameters:	MSG_B0_LM_READ	*message_P
Returns:	0	

2.5.2.16 b0_model_directory.c

The b0_model_directory function a stub for future enhancement; it is not currently implemented.

The function call is **b0_model_directory()**. The function always returns 0.

2.5.2.17 b0_model_entry.c

The b0_model_entry function processes the message MSG_B0_MODEL_ENTRY for Ballistics. This message is sent by download_bvols to add entries to the terrain or DED model directory.

The function call is **b0_model_entry(message_P)**, where *message_P* is a pointer to the MSG_B0_MODEL_ENTRY message. The function always returns 0.

Called By:	bx_task	
Routines Called:	BCOPY	
Parameters:	MSG_B0_MODEL_ENTRY	*message_P
Returns:	0	

2.5.2.18 b0_new_frame.c

The b0_new_frame function processes the message MSG_B0_NEW_FRAME for Ballistics. simulation passes this message to give Ballistics new frame information (frame count and the new state of all dynamic models). b0_new_frame then processes each active round.

The function call is **b0_new_frame(message_P)**, where *message_P* is a pointer to the MSG_B0_NEW_FRAME message. The function always returns 0.

When it is called, b0_new_frame processes each active round as follows:

- Calls bx_trajectory to see where the round's trajectory ends.
  - If the trajectory extends beyond the viewing space, b0_new_frame sends a MISS message, then deletes the round.
  - If the trajectory ends within the viewing space, b0_new_frame calls bx_chord_intersect to determine what was hit, returns a HIT_RETURN message, then deletes the round.
- For rounds that are to be traced, b0_new_frame calculates the position and returns a ROUND_POSITION message.

Called By:	bx_task	
Routines Called:	bx_chord_intersect bx_trajectory DELETE_ROUND GET_LB_FROM_LM mx_push	
Parameters:	MSG_B0_NEW_FRAME	*message_P
Returns:	0	

2.5.2.19 b0_print.c

The b0_print function is a generalized message printing routine. The message is printed to stdout.

The function call is **b0_print(message_P)**, where *message_P* is a pointer to the message. The function always returns 0.

Called By:	bx_task	
Routines Called:	printf puts	(if running on a master board) (if running on a slave board)

Parameters:	char	*message_P
Returns:	0	

2.5.2.20 b0_process_chord.c

The b0_process_chord function is a stub for future enhancement; it is not currently implemented.

The function call is **b0_process_chord()**. The function always returns 0.

Called By:	none
Routines Called:	none
Parameters:	none
Returns:	0

2.5.2.21 b0_process_round.c

The b0_process_round function processes the message MSG_B0_PROCESS_ROUND. This message is sent by simulation upon request from the Simulation Host. The message specifies the round id, database id, round type, tracer type, frame rate, mode, proximity range, gun's position and velocity, and gun's elevation and azimuth.

The function call is **b0_process_round(message_P)**, where *message_P* is a pointer to the MSG_B0_PROCESS_ROUND message.

b0_process_round does the following:

- Validates the round type.
- Calls NEW_ROUND to get a round from the free list and put in on the active list.
- Verifies that the gun barrel is within active area memory; deletes the round if it is not.
- Calls bx_trajectory to see if the round's trajectory exceeds active area memory; returns a MISS message and deletes the round if it does.
- Calls bx_chord_intersect to see what the round hit; returns a HIT_RETURN message and deletes the round.
- For rounds that are to be traced, calculates the position and returns a ROUND_POSITION message.

The function returns 0 if successful. It returns -1 if the round fired is not of a known type, the free list is empty (i.e., the maximum number of active rounds has been reached), or the gun barrel is not within the AAM database.

Called By:	bx_task
Routines Called:	bx_chord_intersect bx_trajectory DELETE_ROUND GET_LB_FROM_LM mx_push NEW_ROUND
Parameters:	MSG_B0_PROCESS_ROUND *message_P
Returns:	0 -i

2.5.2.22 b0_round_fired.c

The b0_round_fired function processes the message MSG_BC_ROUND_FIRED for Ballistics. This message is sent by simulation upon request from the Simulation Host. The message specifies the round type, whether or not tracer effects are to be displayed, the round identifier, the gun tip position and velocity, the gun's elevation and azimuth, the estimated time to impact, and the estimated range of impact.

The function call is **b0_round_fired(round_fired_P)**, where *round_fired_P* is a pointer to MSG_B0_ROUND_FIRED the message.

b0_round_fired does the following:

- Validates the round type.
- Calls NEW_ROUND to get a round from the free list and put it on the active list.
- Verifies that the gun barrel is within active area memory; deletes the round if it is not.
- Calls bx_trajectory to see if the round's trajectory exceeds active area memory; returns a MISS message and deletes the round if it does.
- Calls bx_chord_intersect to see what the round hit; returns a HIT_RETURN message and deletes the round.
- For rounds that are to be traced, calculates the position and returns a ROUND_POSITION message.

The function returns 0 if successful. It returns -1 if the round fired is not of a known type, the free list is empty, or the gun barrel is outside active area memory.

The MSG_ROUND_FIRED message has been replaced by the MSG_PROCESS_ROUND message. MSG_ROUND_FIRED is retained for backwards compatibility.

Called By:	bx_task

Routines Called:	bx_chord_intersect bx_trajectory DELETE_ROUND GET_LB_FROM_LM mx_push NEW_ROUND	
Parameters:	MSG_B0_ROUND_FIRED	*round_fired_P
Returns:	0 -1	

2.5.2.23 b0_state_control.c

The b0_state_control function processes the message MSG_B0_STATE_CONTROL for Ballistics. simulation uses this message to reset Ballistics or put it into the run state.

The function call is **b0_state_control(message_P)**, where *message_P* is a pointer to the MSG_B0_STATE_CONTROL message.

b0_state_control sets the Ballistics global variable *G_bal_state* to the new state provided. If the new state is BX_RESET, b0_state_control calls bx_reset.

The function always returns 0.

Called By:	bx_task	
Routines Called:	bx_reset	
Parameters:	MSG_B0_STATE_CONTROL	*message_P
Returns:	0	

2.5.2.24 b0_status_request.c

The b0_status_request function is a stub for future enhancement; it is not currently implemented.

The function call is **b0_status_request()**. The function always returns 0.

2.5.2.25 b0_traj_chord.c

The b0_traj_chord function processes the message MSG_B0_TRAJ_CHORD for Ballistics. This message is sent by simulation upon request from the Simulation Host. The message message specifies the tracer effect type, whether or not tracer effects are to be displayed, the chord identifier, and the chord's starting and ending positions (x, y, and z coordinates). This message is also sent by simulation when processing the simulated vehicle's AGL (altitude above ground level).

The function call is **b0_traj_chord(message_P)**, where *message_P* is a pointer to the MSG_B0_TRAJ_CHORD message.

b0_traj_chord does the following:

- Locates the chord in the terrain.
- Calls bx_chord_intersect to determine whether the chord hits anything in the local terrain.
- Pushes either a hit or a miss message (as appropriate) onto the Ballistics message queue.

The function always returns 0.

Called By:	bx_task	
Routines Called:	bx_chord_intersect GET_DB_POS mx_push	
Parameters:	MSG_B0_TRAJ_CHORD	*message_P
Returns:	0	

2.5.2.26 b0_traj_entry.c

The b0_traj_entry function processes the message MSG_B0_TRAJ_ENTRY for Ballistics. This message is used to add entries to a trajectory table. The message is sent by db_mcc_setup when processing a MSG_TRAJ_TABLE_XFER message from the Simulation Host.

The function call is **b0_traj_entry(message_P)**, where *message_P* is a pointer to the MSG_B0_TRAJ_ENTRY message.

The function returns 0 if successful. It returns -1 if the trajectory type is invalid. It returns 1 if the trajectory table is full.

Called By:	bx_task	
Routines Called:	outhexl puts	(if running on a slave board) (if running on a slave board)
Parameters:	MSG_B0_TRAJ_ENTRY	*message_P

Returns:	1 0 -1

2.5.2.27 b0_undefined_message.c

The b0_undefined_message function outputs the "*** Undefined Message ***" error for slave Ballistics boards.

The function call is **b0_undefined_message()**. The function always returns 0.

Called By:	bx_task
Routines Called:	puts (if running on a slave board)
Parameters:	none
Returns:	0

2.5.3 Ballistics Intersection Calculations

This section details the CSUs in Ballistics Intersection Calculations component of the Ballistics Processing CSC. It contains the functions that are responsible for calculating chord intersections (hits) for various purposes.

The driving function is bx_chord_intersect. This function is called by the functions in the Ballistics Interface Message Processing component that deal with processing rounds or tracing trajectories. bx_chord_intersect calls other Ballistics Intersection Calculations functions to check for intersections with various objects (static vehicles, dynamic vehicles, terrain bvols, and terrain polygons).

2.5.3.1 bx_bvol_int.c

The bx_bvol_int function intersects a chord with a bounding volume. This function is called by bx_chord_intersect to check for intersections with terrain bounding volumes, and is called by bx_model_int to check for intersections with model (vehicle) bounding volumes.

The function call is **bx_bvol_int(start, end, pbvl, ratio_to_intersect, vehicle_flag)**, where:

start is the chord's starting point
end is a pointer to the return location for the chord's ending point (the intersection point); returned by bx_bvol_int
pbvl is a pointer to the bvol entry
ratio_to_intersect is a pointer to the return location for the distance from the chord's start point to the intersection point, divided by the total length of the chord; this value is returned by bx_bvol_int and is useful when transforming chord points into the vehicle coordinate system
vehicle_flag is TRUE if the model is a vehicle, FALSE if not

bx_bvol_int does the following:

- Checks the bvol's vertices against the chord's start and end points to see if they intersect. Returns FALSE if they do not.
- Clips backfaces (the sides of a polygon that face away from the viewpoint).
- Checks for start- and endpoints on the same side of the bounding volume.
- Checks for hits on the top or bottom of the bounding volume.
- Clips around the quadrilateral projection of the bounding volume.
- Sets the chord's ending position.

The function returns 1 if successful or 0 if no intersection is detected. The function also returns the intersection point and the ratio_to_intersect by placing the data in the locations specified in the call.

Called By: bx_chord_intersect
bx_model_int

Routines Called: none

Parameters:

R4P3D	*start
R4P3D	*end
BVOL_ENTRY	*pbvl
REAL_4	*ratio_to_intersect
BOOLEAN	vehicle_flag

Returns: 1 (TRUE)
0 (FALSE)

2.5.3.2 bx_chord_intersect.c

The bx_chord_intersect function determines whether a given chord intersects with anything in active area memory. It calls other functions in the Ballistics Intersection Calculations component to check for intersections with models or the terrain, then creates the hit or miss message.

The function call is **bx_chord_intersect(chord_P, buffer_P, aam_index, dv_ex_flag, dv_veh_id)**, where:

chord_P is a pointer to the chord's data
buffer_P is a pointer to the hit return data
aam_index is the AAM partition index
dv_ex_flag is TRUE if a particular vehicle is to be excluded from intersection processing, or FALSE if all vehicles are to be included
dv_veh_id is the id of the vehicle to be excluded, if *dv_ex_flag* is TRUE

bx_chord_intersect does the following:

- Checks for hits on pre- and post-processed dynamic models.
- Calls bx_get_lm_grid to find the load modules to be searched, based on the chord's location.
- Calls bx_model_int to check for intersections with static models.
- Calls bx_model_int to check for intersections with dynamic models.
- Calls bx_get_lm_data to get data for the load module (if not in cache).
- Calls bx_bvol_int to check for intersections with terrain bounding volumes.
- Calls bx_poly_int to check for intersections with terrain polygons.
- Builds the hit return message (to be returned to simulation by the calling routine).

The function returns 1 if an intersection is detected. It returns 0 if no intersection was detected, or if the load module could not be found.

Called By: b0_new_frame
b0_process_round
b0_round_fired
b0_traj_chord

Routines Called: BCOPY

bx_bvol_int
bx_get_lm_data
bx_get_lm_grid
bx_model_int
bx_poly_int
GET_LB_FROM_LM

Parameters:		
	CHORD	*chord_P
	BYTE	*buffer_P
	HWORD	aam_index
	BOOLEAN	dv_ex_flag
	HWORD	dv_veh_id

Returns: 1 (TRUE)
0 (FALSE)

2.5.3.3 bx_functions.c

The bx_functions.c CSU contains utility functions used for Ballistics. These functions are the following:

- bx_new_round
- bx_delete_round
- bx_get_db_pos
- bx_get_chord_end
- bx_new_bvol
- bx_free_lm_cache
- bx_new_poly
- bx_get_lb_from_lm
- bx_new_stat_veh
- bx_delete_stat_veh
- bx_dist_sq_pt_line

Note: *Most of these functions are no longer used. Macros (see Appendix B) are used instead, to increase performance.*

2.5.3.3.1 bx_new_round

The bx_new_round function gets a new round from the free list, and increments the number of active rounds. The function returns a pointer (*new_round_P*) to the new round. The pointer is set to NULL if no free rounds are available.

The function call is **bx_new_round()**.

This function is not currently used. The NEW_ROUND macro is used to get rounds from the free list.

Called By: none

Routines Called:	none
Parameters:	none
Returns:	new_round_P

2.5.3.3.2 bx_delete_round

The bx_delete_round function removes a round from the active list and puts it on the free list. It then decrements the number of active rounds and increments the number of free rounds.

The function call is **bx_delete_round(dead_round_P)**, where *dead_round_P* is a pointer to the round to be deleted.

This function is not currently used. The DELETE_ROUND macro is used to delete active rounds.

Called By:	none	
Routines Called:	none	
Parameters:	ROUND_DATA	*dead_round_P
Returns:	none	

2.5.3.3.3 bx_get_db_pos

The bx_get_db_pos function finds the load module that corresponds to a given point in the database.

The function call is **bx_get_db_pos(point_P, lm_width, inv_lm_width, lm_per_side)**, where:

point_P is a pointer to the location in the database
lm_width is the width of a load module
inv_lm_width is the inverse of the width of a load module
lm_per_side is the number of load modules in a row or column of AAM (usually 16)

This function is not currently used. The GET_DB_POS macro is used to find database positions.

Called By:	none

Routines Called:	FIND_LM	
Parameters:	POINT_DATA HWORD REAL_4 HWORD	*point_P lm_width inv_lm_width lm_per_side
Returns:	none	

2.5.3.3.4 bx_get_chord_end

The bx_get_chord_end function finds the end of the current chord (and, therefore, the beginning of the next chord in the trajectory), given an active round and a trajectory table entry.

The function call is **bx_get_chord_end(chord_P, round_message_P, traj_entry_P, offset)**, where:

chord_P is a pointer to the chord
round_message_P is a pointer to the active round
traj_entry_P is is a point to the trajectory table entry
offset is the gun barrel velocity offset

This function is not currently used.

Called By:	none	
Routines Called:	none	
Parameters:	CHORD MSG_B0_PROCESS_ROUND TRAJ_ENTRY REAL_4	*chord_P *round_message_P *traj_entry_P offset
Returns:	none	

2.5.3.3.5 bx_new_bvol

The bx_new_bvol function gets a new bounding volume from the free list and adds it to a load module list. If there are no free bvols, bx_new_bvol swaps out the least-recently-used load module.

The function call is **bx_new_bvol(lm_dir)**, where *lm_dir* is a load module in the cache.

The function returns a pointer (*bvol_P*) to the new bounding volume.

Called By:	bx_get_lm_data	
Routines Called:	FREE_LM_CACHE	
Parameters:	LM_CACHE_ENTRY	*lm_dir
Returns:	bvol_P	

2.5.3.3.6 bx_free_lm_cache

The bx_free_lm_cache function, when given a load module in the Ballistics database cache, puts the bounding volumes in that module on the free bvol list, and puts the polygons in that module on the free polygon list.

The function call is **bx_free_lm_cache(lm_dir)**, where *lm_dir* is a load module in the cache.

This function is not currently used. The FREE_LM_CACHE macro is used to free load module bvols and polygons.

Called By:	*none*	
Routines Called:	none	
Parameters:	LM_CACHE_ENTRY	*lm_dir
Returns:	none	

2.5.3.3.7 bx_new_poly

The bx_new_poly function gets a new polygon from the free list and puts it on a specified load module list. If there are no free polygons, bx_new_poly swaps out the least-recently-used load module.

The function call is **bx_new_poly(lm_dir)**, where *lm_dir* is a load module in the cache.

The function returns a pointer to the new polygon.

Called By:	bx_get_lm_data
Routines Called:	FREE_LM_CACHE

Parameters:	LM_CACHE_ENTRY	*lm_dir
Returns:	poly_P	

2.5.3.3.8 bx_get_lb_from_lm

The bx_get_lb_from_lm function takes a load module number and returns the number (0 to 255) of the load block that module is in.

The function call is **bx_get_lb_from_lm (lm)**, where *lm* is the load module number (0 to 1023).

This function is not currently used. The GET_LB_FROM_LM macro is used to determine load block numbers.

Called By:	none	
Routines Called:	none	
Parameters:	INT_4	lm
Returns:	row*16 + column	

2.5.3.3.9 bx_new_stat_veh

The bx_new_stat_veh function gets a static vehicle from the free list and adds it to the list of the specified load module.

The function call is **bx_new_stat_veh(veh_table_P)** where *veh_table_P* is a pointer to the vehicle table.

The function returns a pointer to the new static vehicle. It returns NULL if no pointers are available (i.e., the maximum number of static vehicles has been reached).

This function is not currently used. The NEW_STAT_VEH macro is used to put a static vehicle into a load module's list.

Called By:	none	
Routines Called:	none	
Parameters:	STRUCT_P_SV	*veh_table_P

Returns:	NULL new_sv_P	

2.5.3.3.10 bx_delete_stat_veh

The bx_delete_stat_veh function removes a static vehicle from a specified load module list and returns it to the free list.

The function call is **bx_delete_stat_veh(dead_sv_P, table_P)**, where:

dead_sv_P is a pointer to the static vehicle to be deleted
table_P is a pointer to the vehicle table

This function is not currently used. The DELETE_STAT_VEH macro is used to delete static vehicles.

Called By:	none	
Routines Called:	none	
Parameters:	STAT_VEH STRUCT_P_SV	*dead_sv_P *table_P
Returns:	none	

2.5.3.3.11 bx_dist_sq_pt_line

The bx_dist_sq_pt_line function finds the distance squared between a point and a line segment.

The function call is **bx_dist_sq_pt_line(pt_P, start_P, end_P)**, where:

pt_P is a pointer to the point
start_P is a pointer to the start of the line segment
end_P is a pointer to the end of the line segment

The function returns the result of the calculation as *result*. It returns 1000000.00 if the result is less than 0.

Called By:	bx_model_int
Routines Called:	none

Parameters:	R4P3D	*pt_P
	R4P3D	*start_P
	R4P3D	*end_P

Returns:	1000000.00 result

2.5.3.4 bx_get_lm_data.c

The bx_get_lm_data function finds and caches all bounding volumes and polygons in a given load module that have their local terrain or Ballistics bit set to true. The function can also be used to cache all bvols and polygons in the load module, regardless of their local terrain and Ballistics bits. This function is called by bx_chord_intersect to get load module data from the AAM if it is not already cached.

The function call is **bx_get_lm_data(lm_addr, lm_dir, poly_mask)**, where:

lm_addr is the address of the load module
lm_dir is the load module directory
poly_mask is TRUE if all polygons are to be cached, regardless of the state of their local terrain and Ballistics bits

The function always returns 0.

Called By:	bx_chord_intersect	
Routines Called:	bx_new_bvol	
	bx_new_poly	
	FXTO881	
Parameters:	WORD	lm_addr
	LM_CACHE_ENTRY	*lm_dir
	WORD	poly_mask
Returns:	0	

2.5.3.5 bx_get_lm_grid.c

The bx_get_lm_grid function finds the load modules and grids in the database that are intersected by a given chord. It is called by bx_chord_intersect when it is looking for the load modules to search.

The function call is **bx_get_lm_grid(pcrd, lm_per_side, bal_search, dvl_search, lm_width, lm_addr_table)**, where:

pcrd is a pointer to the chord
lm_per_side is the number of load modules in a row or column of AAM

bal_search is used to store load module offsets and grid words
dvl_search is used to store dynamic module path info
lm_width is the width of a load module
lm_addr_table is an array of load module addresses

The function returns 1 if successful. It returns 0 if the chord crosses four load modules, yet one of the grids is not a corner grid of a load module; this is an error condition.

Called By:	bx_chord_intersect	
Routines Called:	none	
Parameters:	CHORD	*pcrd
	HWORD	lm_per_side
	LM_SEARCH_LIST	bal_search[]
	HWORD	dvl_search[]
	HWORD	lm_width
	WORD	lm_addr_table[]
Returns:	1 (TRUE) 0 (FALSE)	

2.5.3.6 bx_model_int.c

The bx_model_int function intersects a chord with a model. This function is called by bx_chord_intersect to check for intersections with static and dynamic vehicles.

The function call is **bx_model_int(chord_P, model_inst_P, hit_data_P)**, where:

chord_P is a pointer to the chord
model_inst_P is a pointer to the model
hit_data_P is a pointer to the data for the hit return message

bx_model_int does the following:

- Based on the model's radius, checks to see if the chord falls completely outside of the model. Returns FALSE if it does.
- Checks the model's first component for a hit.
    - Converts the chord to vehicle coordinates.
    - Translates and rotates the chord.
    - Calls bx_bvol_int to check for a bounding volume intersection. If an intersection is found, sets *hit_flag* to TRUE. Subtracts a fixed offset (*INTERSECT_OFFSET*, currently defined as 1.5%) from the actual *ratio_to_intersect* value. This moves the intersection point slightly away from the middle of the object enclosed by the intersected bvol, causing any special effects for the hit to appear largely outside of the object. Places the hit information in *hit_data_P*.
- If no hit was found, checks the model's second component, if it has one.
    - Rotates the chord into turret coordinates.

- Calls bx_bvol_int to check for a bounding volume intersection. If an intersection is found, sets *hit_flag* to TRUE; subtracts INTERSECT_OFFSET from the *ratio_to_intersect* value; places the hit information in *hit_data_P*.

The function returns *hit_flag* set to TRUE if a hit is detected, or FALSE if no intersection is detected.

Called By:	bx_chord_intersect	
Routines Called:	bx_bvol_int	
Parameters:	CHORD	*chord_P
	STAT_VEH	*model_inst_P
	MSG_B1_HIT_RETURN	*hit_data_P
Returns:	hit_flag	

2.5.3.7 bx_poly_int.c

The bx_poly_int function intersects a chord and a polygon. This function is called by bx_chord_intersect to check for intersections with terrain polygons.

The function call is **bx_poly_int(start, end, vtx_count, pvtx)**, where:

start is the starting point of the chord
end is a pointer to the return location for the ending point of the chord (the point of intersection)
vtx_count is the number of vertices in the polygon
pvtx is a pointer to the polygon vertex data

bx_poly_int does the following:

- Clips around the polygon using the minimum and maximum values and a fixed offset (currently set at 10 meters).
- Makes the polygon normals.
- Calculates the cross product.
- Clips out backface intersections.
- Checks to see if the intersection is in the interior of the polygon.
- Finds the normal-to-polygon side by taking the cross product of the polygon normal and the polygon side.

The function returns 1 if the chord intersects the polygon, or 0 if it does not. The intersection point is placed in the *end* location specified in the call.

Called By:	bx_chord_intersect

Routines Called:	none	
Parameters:	WORD R4P3D R4P3D R4P3D	vtx_count *start *end *pvtx[]
Returns:	1 (TRUE) 0 (FALSE)	

2.5.3.8 bx_reset.c

The bx_reset function resets Ballistics. bx_reset is called by b0_state_control when the message from simulation specifies a new state of BX_RESET.

The function call is **bx_reset()**. bx_reset reclaims dynamic memory, then initializes the following structures:

- Terrain and dynamic elements database (DED) model directories.
- Terrain and DED bounding volume directories.
- Static vehicle list.
- Bounding volume cache list.
- Polygon cache list.
- Round list.
- Trajectory table directory.
- Various pointers, lists, and temporary variables.

Called By:	b0_state_control
Routines Called:	free free133
Parameters:	none
Returns:	none

2.5.3.9 bx_trajectory.c

The bx_trajectory function returns the position of a projectile using the provided trajectory tables.

The function call is **bx_trajectory(round_P)**, where *round_P* is a point to the round data. bx_trajectory does the following:

- If this is the first call for a new round, finds the trajectory table for the round type.
- Rotates through the elevation angle.

- Rotates through the azimuth angle.
- Adds in the gun position and velocity.

The function returns 1 if it finds the position in the database. It returns 0 if the round travels beyond the viewing space, or if the end of the trajectory table was reached.

Called By:	b0_new_frame b0_process_round b0_round_fired GET_DB_POS	
Routines Called:	none	
Parameters:	ROUND_DATA	*round_P
Returns:	1 (TRUE) 0 (FALSE)	

2.5.4 Ballistics Message Queue Processing

This section details the CSUs in Ballistics Message Queue Processing, a major functional component of the Ballistics Processing CSC. These functions are responsible for manipulating and maintaining the queues that make up the interface between Ballistics and real-time software.

2.5.4.1 mx_error.c

The mx_error function returns a Ballistics error message. The function is called by bx_task to provide a text message for output to the operator.

The function call is **mx_error(status)**, where *status* is the error message.

Called By:	bx_task download_bvols simulation upstart	
Routines Called:	none	
Parameters:	WORD	status
Returns:	"DEVICE CLOSED" "DEVICE TABLE FULL" "DEVICE OPENED" "DEVICE BUSY" "DEVICE EMPTY" "DEVICE FULL" "MESSAGE PUSHED" "MESSAGE POPPED" "MESSAGE PREVIEWED" "MESSAGE SKIPPED" "UNDEFINED ERROR" "UNDEFINED RETURN"	

2.5.4.2 mx_open.c

The mx_open function opens an MX device over a queue message.

The function call is **mx_open(dev_P, device_size)**, where:

dev_P is a pointer to the MX device (message queue)
device_size is the size of the message queue

The function always returns MX_DEVICE_OPENED.

Called By:	bx_task upstart	
Routines Called:	sc_lock sc_unlock	
Parameters:	MX_DEVICE INT_4	*dev_P device_size
Returns:	MX_DEVICE_OPENED	

2.5.4.3 mx_peek.c

The mx_peek function previews a queue message.

The function call is **mx_peek(dev_P, message_code, message_size, message_addr)**, where:

dev_P is a pointer to the message queue
message_code is the message type
message_size is the size of the message
message_addr is a pointer to the return location for a pointer to the message's address

If successful, the function returns MX_MESSAGE_PREVIEWED and places a pointer to the message at the head of the queue in the *message_addr* location specified in the call. The function returns MX_DEVICE_EMPTY if the specified queue contains no messages.

Called By:	bx_task simulation upstart	
Routines Called:	sc_lock sc_unlock	
Parameters:	MX_DEVICE HWORD HWORD BYTE	*dev_P *message_code *message_size **message_addr
Returns:	MX_DEVICE_EMPTY MX_MESSAGE_PREVIEWED	

2.5.4.4 mx_push.c

The mx_push function pushes a message onto the Ballistics message queue.

The function call is **mx_push(dev_P, source_address, message_code, message_size)**, where:

dev_P is a pointer to the message queue
source_address is the address of the message
message_code is the type of message
message_size is the number of bytes in the message

The function returns MX_MESSAGE_PUSHED if successful. It returns MX_DEVICE_FULL if the specified message queue is already full.

Called By:	b0_new_frame b0_process_round b0_round_fired b0_traj_chord bx_task db_mcc_setup download_bvols getside open_dbase rowcol_rd simulation	
Routines Called:	BCOPY sc_lock sc_unlock	
Parameters:	MX_DEVICE WORD HWORD HWORD	*dev_P source_address message_code message_size
Returns:	MX_DEVICE_FULL MX_MESSAGE_PUSHED	

2.5.4.5 mx_skip.c

The mx_skip function skips over a message in the queue. The message at the head of the queue is flushed, and the next message moves to the head of the queue.

The function call is **mx_skip(dev_P)**, where *dev_P* is a pointer to the queue.

The function returns MX_MESSAGE_SKIPPED if successful. It returns MX_DEVICE_EMPTY if the specified message queue contains no messages.

Called By:	bx_task simulation upstart	
Routines Called:	sc_lock sc_unlock	
Parameters:	MX_DEVICE	*dev_P
Returns:	MX_DEVICE_EMPTY MX_MESSAGE_SKIPPED	

2.5.4.6 mx_wcopy.c

The mx_wcopy function performs a block copy.

The function call is **mx_wcopy (source_P, destination_P, byte_count)**, where:

source_P is a pointer to the source data
destination_P is a pointer to the destination location
byte_count is the number of bytes to be copied

This function is not currently used.

Called By:	none	
Routines Called:	none	
Parameters:	WORD WORD INT_2	*source_P *destination_P byte_count
Returns:	none	

2.6 User Interface (GOSSIP) CSC

This section describes the functions that make up the Gossip CSC. This CSC provides a backdoor user interface which allows various debugging and query features during runtime operation. Gossip provides the ability to interrogate system performance, view and modify system memory, and debug real-time problems.

The Gossip user can do the following:

- Display data from the Ballistics database.
- Display data from the terrain and DED databases.
- Display DR11 variables.
- Initiate and run demos.
- Initiate and use flying mode.
- Initiate and interface with Flea (the Simulation Host emulator).
- Display current information about simulation memory.
- Modify simulation memory.
- Display static and dynamic models.
- Invoke a DTP emulator.
- Interface to the 2-D overlay processor (120TX systems only).
- Perform calibration acceptance tests (120TX systems only).
- Load color polygons.
- Display and change various system variables.
- Display DR11 message packets.
- Enable and disable frame interrupts.
- Enable and disable single-step mode.
- Place a calibration pattern on all channels.
- Change the default database or configuration file.
- Start, stop, or reset timers,

The gossip task runs at the lowest priority, to prevent interference with the simulation.

The CSUs contained in the Gossip CSC are identified in Figure 2-15 and described in this section.

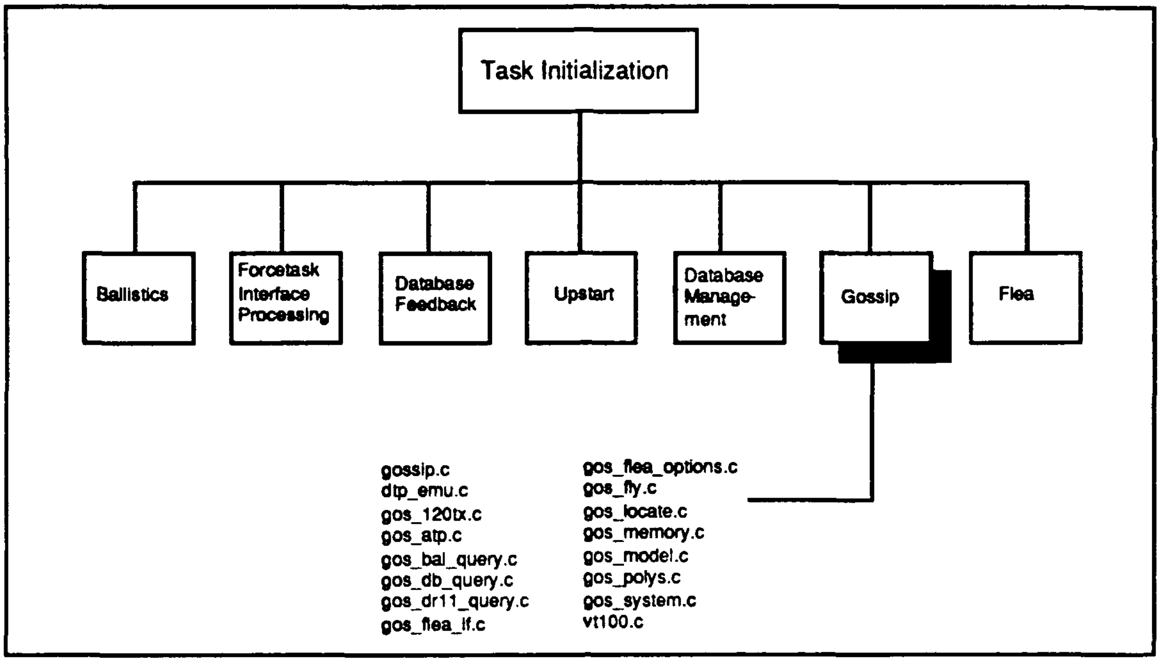


Figure 2-15. Gossip CSUs

Figure 2-16 illustrates the interaction between the major CSUs in Gossip.

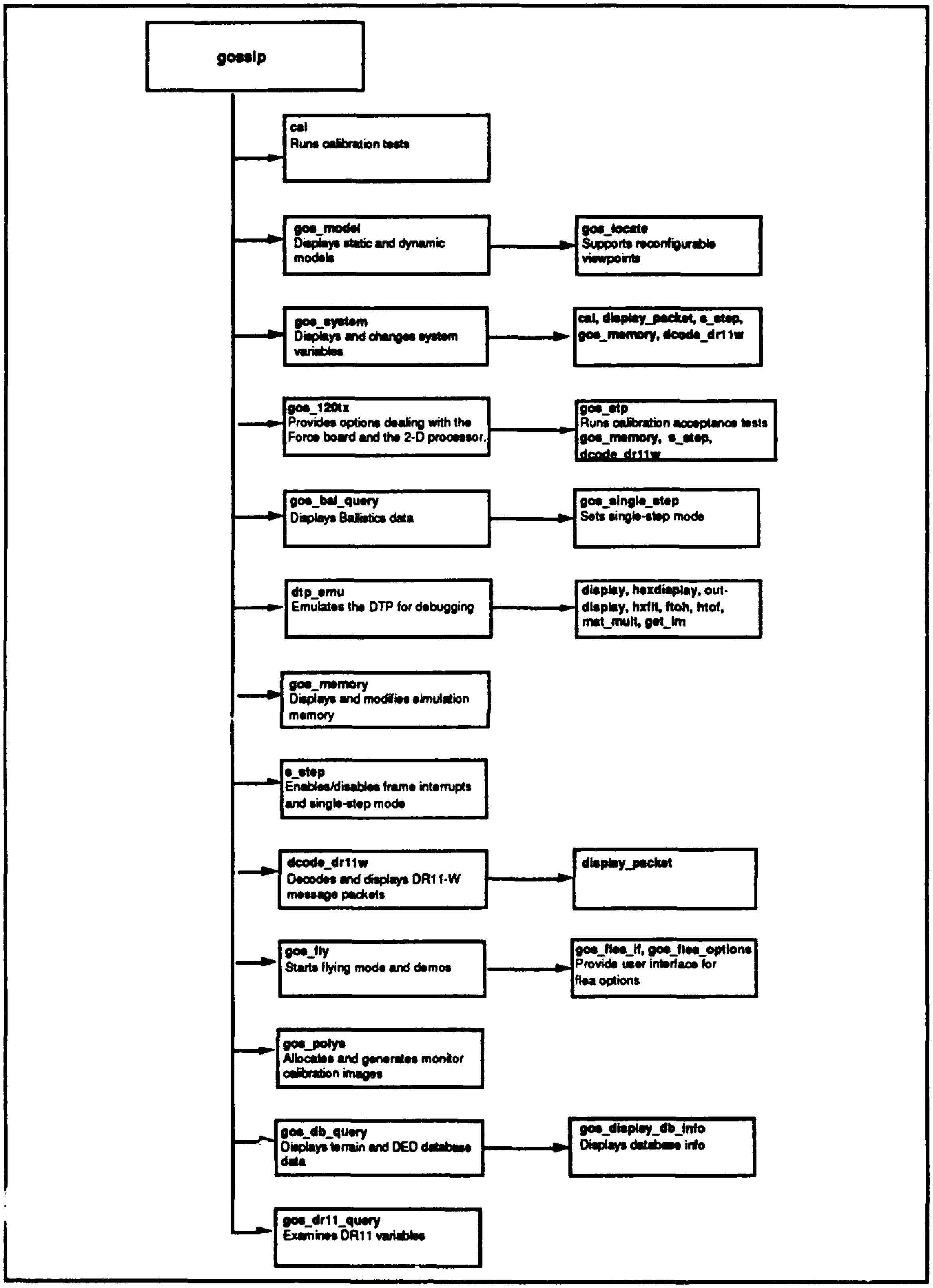


Figure 2-16. Gossip Flow Diagram

2.6.1 dtp_emu.c

The dtp_emu.c CSU contains the functions used to emulate the Data Traversal Processor (DTP) for debugging. These functions are the following:

- dtp_emu
- display
- outdisplay
- hxflt
- hexdisplay
- ftoh
- htof
- mat_mult
- get_lm

2.6.1.1 dtp_emu

The dtp_emu function is a DTP emulator used in debugging. The function is invoked from gossip when the user selects the "dtp emulator" option from the Gossip main menu. The DTP is a micro-coded processor board that sends data to the Polygon Graphics Processor, based on commands placed in active area memory by the DTP Command Generator. dtp_emu emulates the functions performed by the DTP.

The function call is **dtp_emu()**. Once dtp_emu is invoked, the Gossip user can request the following:

- Set poly data display mode on or off.
- Set the display mode to float or hex.
- Set tracing on or off.
- Set system interrupts on or off.
- Display the current modes (display, poly data, system interrupt, and trace) and the DTP stack pointer.
- Display the DTP stack
- Start the DTP emulator.
- Step through the various DTP commands.
- Restart the emulator.
- Set the memory address for the emulator program counter.
- Set the address of the AAM peek (view) register.
- Set the address of the emulator peek (view) register.
- Write the contents of AAM.
- Set break points (currently not implemented).

Called By:	gossip
Routines Called:	display ftoh get_lm hexdisplay htof

hxflt
mat_mult
outdisplay
printf
scanf
sqrt
sysrup_off
sysrup_on
unbf_getchar
XCLOSE
XLSEEK
XOPEN
XREAD

Parameters: none

Returns: none

2.6.1.2 display

The display function is used to convert hexadecimal digits or floating point numbers for display purposes.

The function call is **display(ptr, num, poly)**, where:

ptr is a pointer to the data in AAM
num is the number of characters to convert
poly is **LOAD** if a load module is being processed, or **POLY** if a polygon is being processed

The function always returns 1.

Called By: dtp_emu

Routines Called: hxflt
printf

Parameters:	INT_4	**ptr
	INT_2	num
	INT_2	poly

Returns: 1

2.6.1.3 outdisplay

The outdisplay function is used to display formatted data depicting polygon commands in the DTP processing path.

The function call is **outdisplay(ptr, wd_count)**, where:

ptr is the AAM pointer to the start of the Poly Processor command
wd_count is the number of bytes in the command

The function returns 0 if successful or 1 if the command could not be displayed.

Called By:	dtp_emu	
Routines Called:	hxflt printf	
Parameters:	INT_4 WORD	**ptr wd_count
Returns:	0 1	

2.6.1.4 hxflt

The hxflt function is used to convert hexadecimal characters for output to the display.

The function call is **hxflt(h)**, where *h* is the character to be converted.

Called By:	dtp_emu outdisplay	
Routines Called:	htof printf	
Parameters:	WORD	h
Returns:	none	

2.6.1.5 hexdisplay

The hexdisplay function is used to display hexadecimal numbers.

The function call is **hexdisplay(pntr, args)**, where:

pntr is the AAM address of the data to be displayed
args is the number of digits to display

Called By:	dtp_emu	
Routines Called:	printf	
Parameters:	INT_4 INT_4	**pntr args
Returns:	none	

2.6.1.6 ftoh

The ftoh function is used to convert an IEEE floating point value to internal hex representation for display.

The function call is **ftoh(f, h)**, where:

f is the floating point value
h is the hexadecimal equivalent

Called By:	dtp_emu mat_mult	
Routines Called:	none	
Parameters:	REAL_4 WORD	*f *h
Returns:	*h	

2.6.1.7 htof

The htof function is used to convert a hexadecimal number to IEEE floating point for display.

The function call is **htof(h, f)**, where:

h is the hexadecimal value
f is the floating point equivalent

Called By:	dtp_emu hxflt mat_mult	
Routines Called:	none	
Parameters:	WORD REAL_4	*h *f
Returns:	*f	

2.6.1.8 mat_mult

The mat_mult function is used to multiply (concatenate) two matrices to generate a third matrix.

The function call is **mat_mult(a, b, c)**, where:

a is the address of the first matrix
b is the address of the second matrix
c is the address of the result matrix

Called By:	dtp_emu	
Routines Called:	ftoh htof printf	
Parameters:	WORD WORD WORD	*a *b *c
Returns:	none	

2.6.1.9 get_lm

The get_lm function is used to simulate the DTP function of getting the next load module pointer for processing.

The function call is **get_lm(flag)**, where *flag* is **0** (open -> hdglut -> lmlut), **1** (lmlut), **2** (close), or **3** (hdglut -> lmlut).

The function returns 1 if successful, or 0 if an error occurred.

Called By:	dtp_emu	
Routines Called:	printf XCLOSE XLSEEK XOPEN XREAD	
Parameters:	INT_2	flag
Returns:	0 1	

2.6.2 gos_120tx.c

The gos_120tx function provides options to the Gossip user that are available only on a 120TX CIG. These options all deal with 2-D overlays and the Force board. This function is invoked by gossip when the user selects the "120tx/t menu" option from the Gossip main menu.

The function call is **gos_120tx()**. The following table identifies the function called or the action taken by gos_120tx for each option on its main menu.

gos_120tx Menu Option		Processing by gos_120tx
1	Start/Stop 2D updates	Sets gsp_io_flag.
2	Enable/Disable Force timers	Sets force_timing_flag.
3	Change look up tables	Prompts user for table code (out the window, daylight TV, white hot, or black hot); sets dtv_therm_word accordingly.
a	Perform acceptance tests	Calls gos_atp.
d	(Does not appear on menu)	Calls dcode_dr11w.
g	Talk to 2D process/mem	See table below.
m	(Does not appear on menu)	Calls gos_memory.
p	Sets pixel depth request i,j	Asks user for pixel i and j positions; shows Force locations.
r	(Does not appear on menu)	Returns pixel depth for pixel i and j.
s	(Does not appear on menu)	Calls s_step.

Selecting the "Talk to 2D process/mem" option (g) displays the FORCE-2D Communications Menu. The following table identifies the function called or the action taken by gos_120tx for each option on this menu.

FORCE-2D Communications Menu Option	Processing by gos_120tx
0 Restart 2d processor	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_NMI_START.
4 Read Host Control	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_READ_HCTRL.
5 Write Host Control	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_WRITE_HCTRL.
6 Read Data	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_READ_HDATA.
7 Write Data	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_WRITE_HDATA.
9 Halt 2D Processor	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_STOP.
a Set GSP address to read/write	Asks user for the GSP address; sets gsp_temp_addr.
b Set number of times to fill mem	Asks user for number of times to fill memory; sets fill_mem_count.
e Send mail to 2D processor	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_MAIL_SEND.
f Display force/2D registers	Displays Front End Control Register, Force Control Register, Force Status Register, Force Errors Register, GSP Address, HWORDS count, Repeat Block Fill Count.
g Read data from 2D processor memory	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_READ_START.
i Start memory fill	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_WRITE_START.
l Load output buffer with pattern o	Asks user for 16-bit pattern; sets SUBSYS_DATA_BUFF_OUT.
m (Does not appear on menu)	Calls gos_memory.
n (Does not appear on menu)	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_NMI_START.
o Load output buffer (16 bits)	Prompts user for data; sets SUBSYS_DATA_BUFF_OUT.
p Write data to 2D processor memory	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_WRITE_START.
r View input data buffer	Displays contents of buffer.
t One time communications test	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_TEST_MEM.
w Set word count to read/write	Asks user for the word count; sets SUBSYS_DATA_COUNT.
y Endless communications test	Calls CHECK_FORCE; sets FE_CONTROL to SUBSYS_TEST_MEM2.

The CHECK_FORCE macro referenced in the above table checks to see if the forcetask is running. If it is, the user is asked to retry later. (This prevents the Gossip operation from interfering with processing required for the simulation.) FE_CONTROL is the front-end control register; the value placed in the register tells the forcetask what command to perform.

Called By: gossip

Routines Called: dcode_dr11w
gos_atp
gos_memory
printf
s_step
scanf
unbf_getchar

Parameters: none

Returns: none

2.6.3 gos_atp.c

The gos_atp function is used to run acceptance tests that use the calibration database. This function is called by gos_120tx when the user selects the "perform acceptance tests" option from its main menu.

The function call is **gos_atp()**. The following calibration tests are available:

- Populated Area
- Depth Complexity
- Color Resolution
- Full Perspective Texture
- Level of Detail
- Moving Models (plant, display)
- Occulting Levels
- Polygon Throughput
- Texture with Transparency
- Polygon Test Pattern

Called By: gos_120tx

Routines Called: gos_memory
printf
sc_post
unbf_getchar

Parameters: none

Returns: none

2.6.4 gos_bal_query.c

The gos_bal_query function displays data from the Ballistics database. This function is invoked from gossip when the user selects the "query ballistics" option from the Gossip main menu.

The function call is **gos_bal_query()**. The function can be used to:

- Set ballistics addresses to user-specified values. (This is required before any other function can be accessed; the addresses can also be changed later on.)
- List any of the following information:
  - ballistics configuration (frame rate and AAM partitions)
  - a user-specified trajectory directory
  - free bvols directory
  - active rounds
  - frame count
  - load module cache information for a user-specified load module
  - load module bounding volumes for a user-specified load module
  - load module cache
  - AAM partition info
  - trajectory table for a user-specified trajectory type
  - free poly directory
  - free rounds directory
  - terrain corners
  - load module polygons for a user-specified load module
- Set single-step mode (by calling gos_single_step).
- Print MSG_PROCESS_ROUND messages.
- Print MSG_TRAJ_CHORD messages.

Called By: gossip

Routines Called: FIND_LM
gos_single_step
PAGE_FORMAT
printf
scanf
unbf_getchar

Parameters: none

Returns: none

2.6.5 gos_db_query.c

The gos_db_query.c CSU is used to examine database information. It contains two functions:

- gos_db_query
- gos_display_db_info

2.6.5.1 gos_db_query

The gos_db_query function examines terrain and DED database information. This function is invoked from gossip when the user selects the "query database" option from the Gossip main menu.

The function call is **gos_db_query()**. The function can be used to do the following:

- Display terrain database information (calls gos_display_db_info).
- Display dynamic elements database information (calls gos_display_db_info).
- List all models.
- List all effects.
- Modify a specified model's component count, process code, or hardware address.
- Modify a specified effect's component count, process code, or hardware address.
- Block copy from a specified source location to a specified destination.

Called By:	gos_model gossip
Routines Called:	gos_display_db_info printf scanf unbf_getchar
Parameters:	none
Returns:	none

2.6.5.2 gos_display_db_info

The gos_display_db_info function is used by gos_db_query to display terrain and dynamic elements database information to the Gossip user.

The function call is **gos_display_db_info(data_P)**, where *data_P* is a pointer to the database header to be displayed.

Called By:	gos_db_query

Routines Called: printf

Parameters: DB_HDR_DBASE_DATA *data_P

Returns: none

2.6.6 gos_dr11_query.c

The gos_dr11_query function examines DR11 variables. This function is invoked from gossip when the user selects the "display dr11 variables" option from the Gossip main menu.

The function call is **gos_dr11_query()**. The function displays the CIG and SIM exchange packet sizes, local terrain chunk size, and local terrain message interval. It then displays the current status of the real-time software: entering data exchange, writing to the Simulation Host, reading from the Simulation Host, or processing messages.

Called By: gossip

Routines Called: printf

Parameters: none

Returns: none

2.6.7 gos_flea_if.c

The gos_flea_if function is used in flying mode and when running demos. gos_flea_if is called by gos_fly if the user requests to enter Flea mode.

The function call is **gos_flea_if()**. The function prompts the user for the viewpoint position and orientation, then posts a FLEA_MB mailbox message to wake up flea. It then waits for a MONITOR_MB mailbox message.

After flea is running, gos_flea_if processes commands to do the following:

- Go forward, go back, stop, change rotation on any axis, change skid on any axis, change velocity, shoot.
- Start, stop, or resume script; display script values.
- Call gos_flea_options if requested by the user.

Called By: gos_fly

Routines Called: blank
cup
gos_flea_options
printf
sc_pend
sc_post
scanf
unbf_getchar

Parameters: none

Returns: none

2.6.8 gos_flea_options.c

The gos_flea_options function displays the Flea options menu, and processes the functions requested by the Gossip user. This function is invoked from gos_flea_if if the user enters # ("flea options") at the Command prompt.

The function call is **gos_flea_options()**. The following actions are supported by gos_flea_options:

- Increase, zero, or decrease velocity.
- Increase, zero, or decrease x, y, or z rotation (to center the steering bar).
- Toggle auto fire.
- Change the round type.
- Add or delete a vehicle.
- Display current location, rotation, AGL, and speed.
- Display hits and misses per minute.
- Plant a static vehicle.
- Remove a static vehicle.
- Fire or process a round.
- Show an effect.
- Show the model list.
- Specify a new process code for a DED model.
- Specify gun overlay data.
- Specify the ammunition define map.

Called By: gos_flea_if

Routines Called: blank
cos
cup
printf
scanf
sin
unbf_getchar

Parameters: none

Returns: none

2.6.9 gos_fly.c

The gos_fly function is used to enter flying mode and to run demos. This function is invoked from gossip when the user selects the "vehicle demo and fly options" option from the Gossip main menu.

The function call is **gos_fly()**. The function lets the Gossip user do the following:

- Start and stop other vehicle demonstrations.
- Start and stop flying in auto-pilot demonstration mode, optionally in endless loop mode. gos_fly posts a SIMULATION_MB message to wake up the simulation function if this option is selected.
- Enter flying mode. gos_fly prompts for the viewpoint position and orientation, then posts a FLEA_MB message to wake up flea. It also provides options to the user to manipulate the vehicle.
- Enter Flea mode. gos_fly calls gos_flea_if.

Called By: gossip

Routines Called: gos_flea_if
printf
sc_post
scanf
unbf_getchar

Parameters: none

Returns: none

2.6.10 gos_locate.c

The gos_locate function traverses the top level of the configuration tree and builds a hull-to-world matrix from the world-to-hull matrix. If the CIG is detected to be supporting simulations of multiple vehicles, gos_locate prompts the Gossip user to identify a reference vehicle.

The function call is **gos_locate(mtx_h_w)**, where *mtx_h_w* is a hull-to-world matrix.

The function returns the hull-to-world matrix if successful. It returns NULL if the configuration tree is not initialized or is empty.

Called By:	gos_model
Routines Called:	printf scanf
Parameters:	REAL_4	*mtx_h_w
Returns:	NULL mtx_h_w

2.6.11 gos_memory.c

The gos_memory function displays relatively current data about simulation memory. This function is invoked from gossip when the user selects the "memory examine/modify" option from the Gossip main menu.

The function call is **gos_memory()**. The function can be used to:

- Display a specified block of memory.
- Modify a specified block of memory.
- Modify a specified memory address.
- Send a snapshot of memory to a specified file.
- Load a snapshot from a specified file into memory.

Called By:	gos_120tx gos_atp gos_model gos_system gossip
Routines Called:	close create_sz open printf read scanf unbf_getchar write
Parameters:	none
Returns:	none

2.6.12 gos_model.c

The gos_model function displays dynamic and static models. This function is invoked from gossip when the user selects the "model menu" option from the Gossip main menu.

The function call is **gos_model()**.

If debug is not enabled, gos_model can be used to do the following:

- Plant a model in tracks.
- Examine memory.

If debug is enabled, the following additional options are supported:

- Add or delete a static vehicle.
- Plant a model.
- Control the DED level of detail (includes moving vehicles and rotating models).
- Select a database for level-of-detail control.
- Database/DED query menu.
- Display effect timing.
- Set the view mode.
- Display view mode.

Called By: gossip

Routines Called:
cos
gos_db_query
gos_locate
gos_memory
model_mtx
printf
rotate_x_nt
rotate_y_nt
rotate_z_nt
scanf
sin
sqrt
sysrup_off
sysrup_on
unbf_getchar

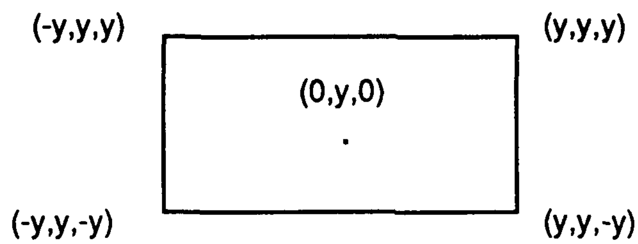
Parameters: none

Returns: none

2.6.13 gos_polys.c

The gos_polys function allocates and generates monitor calibration images. This function is invoked from gossip when the user selects the "load color polygons" option from the Gossip main menu.

The Polygon Processor uses perspective matrices in normalized viewspace (i.e., the field-of-view is not used) when crunching on overlay polygons. The only perspective matrix required for an overlay is a matrix to swap the axes (view space into screen space). The vertices overlay can be described to the Polygon Processor as follows:



where y is the distance from the eye to the overlay.

This means that if the vertices of an overlay (such as the monitor calibration overlay) are given in pixel coordinates, they must be converted to the normalized view space coordinate system. For example, if the screen resolution is 200 x 200, a vertex with pixel coordinates (-50,100) is converted to (-1/2,1).

The function call is **gos_polys()**.

Called By:	gossip
Routines Called:	id_4x3mtx swap_axis
Parameters:	none
Returns:	none

2.6.14 gos_system.c

The gos_system function is used to display and change system variables. This function is invoked from gossip when the user selects the "system status menu" option from the Gossip main menu.

The function call is **gos_system()**. The function can be used to do the following:

- Display local terrain data.

- Display active area data.
- Display the active area map.
- Display a load module header.
- Examine/modify memory — calls gos_memory.
- Set the calibration modifier.
- Print round messages.
- Print chord messages.
- Select hardware display channels.
- Start/stop frame — calls s_step.
- Set the display lights flag.
- Display DR11 message packets — calls dcode_dr11w.
- Change the default database name.

Called By: gossip

Routines Called: cal
dcode_dr11w
display_packet
gos_memory
printf
s_step
scanf
sysrup_off
sysrup_on
unbf_getchar

Parameters: none

Returns: none

2.6.15 gossip.c

The gossip.c CSU contains the functions used to display relatively current data about the simulation. These functions are the following:

- main (for Butterfly compatibility only)
- gossip
- display_packet
- s_step
- dcode_dr11w
- gos_single_step

2.6.15.1 main

The main function is provided for Butterfly compatibility only. It requires one argument: *bvme_id*, which identifies the Butterfly-VME interface. main remaps the addresses used by the Ballistics boards to VME addresses, then calls gossip.

Called By:	none
Routines Called:	Find_Value gossip map_vme printf remap_vme
Parameters:	none
Returns:	none

2.6.15.2 gossip

The gossip function is invoked when Gossip is executed by the user. gossip displays the Gossip main menu, which allows the user to select the type of data to be queried. Depending on the selection made, gossip may prompt for additional information, such as the name of the database or configuration file to use. It then calls the applicable Gossip function.

The following table identifies the function called or the action taken by gossip for each option on the Gossip main menu.

Gossip Main Menu Option	Processing by gossip
1 calibration menu	Calls cal.
2 model menu	Calls gos_model.
3 system status menu	Calls gos_system.
4 120tx/t menu	Calls gos_120tx.
6 dtp emulator	Calls dtp_emu.
b query ballistics	Calls gos_bal_query.
c change default configfile name	Prompts user for new file name; sets global variable.
D display dr11 variables	Calls gos_dr11_query.
d display DR11W messages	Calls dcode_dr11w.
e query database	Calls gos_db_query.
i start/stop dr11w init prints	Toggles dr11w_init_out.
k reset times	*Sets all timers to 0.*
m memory examine/modify	Calls gos_memory.
p load color polygons	Calls gos_polys; calls cal.
s start/stop frame interrupt	Calls s_step.
t start/stop timers	Toggles rtsw_timing_flag.
u change default db name	Prompts user for new database name; sets global variable.
w set DED AAM start address	Prompts user for address; sets global variable.
z vehicle demo and fly options	Calls gos_fly.

Called By: none

Routines Called:
cal
dcode_dr11w
dtp_emu
gos_120tx
gos_bal_query
gos_db_query
gos_dr11_query
gos_fly
gos_memory
gos_model
gos_polys
gos_system
printf
s_step
sc_pend
scanf
strlen
unbf_getchar

Parameters:	INT char	argc *argv
Returns:	none	

2.6.15.3 display_packet

The display_packet function displays the contents of each message in a DR11 exchange packet. This function is called by dcode_dr11w when the user selects the "display DR11W messages" option from the Gossip main menu.

The function call is **display_packet(pntr)**, where *pntr* is a pointer to the message packet.

Called By:	debug_initdr dcode_dr11w gos_system	
Routines Called:	printf	
Parameters:	INT_4	pntr
Returns:	none	

2.6.15.4 s_step

The s_step function is used to (1) enable and disable frame interrupts, and (2) enable and disable single-step mode. This function is called by gossip if the user selects "start/stop frame interrupt" from the Gossip main menu.

The function call is **s_step**(). s_step prompts the user to set/or cancel single-step mode, then does the following:

- If the user requests "interrupts on," s_step calls sysrup_on, then sets *single_step* to FALSE.
- If the user requests "interrupts off," s_step calls sysrup_off, then sets *single_step* to FALSE.
- If the user requests "single-step mode," (used with the "display dr11 variables" option), s_step sets *single_step* to TRUE and *dr11_msg* to TRUE.

Called By:	gos_120tx gos_system gossip
Routines Called:	printf sysrup_on

sysrup_off
unbf_getchar

Parameters: none

Returns: none

2.6.15.5 dcode_dr11w

The dcode_dr11w function decodes and displays DR11 packets. This function is called by gossip if the user selects the "display DR11W messages" option from the Gossip main menu.

The function call is **dcode_dr11w()**. dcode_dr11w calls display_packet to display the input and output packets.

Called By: gos_120tx
gos_system
gossip

Routines Called: display_packet
printf
sysrup_on

Parameters: none

Returns: none

2.6.15.6 gos_single_step

The gos_single_step function forces the system to single-step a real-time frame by posting a message to the simulation mailbox. If gos_single_step detects that *single_step* is TRUE, it calls sysrup_on.

The function call is **gos_single_step()**.

Called By: gos_bal_query

Routines Called: sysrup_on

Parameters: none

Returns: none

2.6.16 vt100.c

The vt100.c CSU contains functions that provide VT100 graphics control. These are:

- cup
- sgr
- double_top
- double_bot
- double_off
- blank
- save_cur
- restore_cur
- scroll_reg

2.6.16.1 cup

The cup function positions the cursor at a specified row and column.

The function call is **cup(r, c)**, where *r* is the row number and *c* is the column number.

Called By: gos_flea_if
gos_flea_options

Routines Called: printf

Parameters: INT_4 r
INT_4 c

Returns: none

2.6.16.2 sgr

The sgr function is used for special graphics renditions.

The function call is **sgr(r)**, where *r* is the row number.

This function is not currently used.

Called By: none

Routines Called: printf

Parameters:	INT_4	r
Returns:	none	

2.6.16.3 double_top

The double_top function represents double-wide, double-high for the top half of the monitor screen.

The function call is **double_top(s)**, where *s* is the starting line.

This function is not currently used.

Called By:	none	
Routines Called:	printf	
Parameters:	BYTE	s
Returns:	none	

2.6.16.4 double_bot

The double_bot function represents double-wide, double-high for the bottom half of the monitor screen.

The function call is **double_bot(s)**, where *s* is the starting line.

This function is not currently used.

Called By:	none	
Routines Called:	printf	
Parameters:	BYTE	s
Returns:	none	

2.6.16.5 double_off

The double_off function returns to single-high and single-width. This reverses the effect of double_top and/or double_bot.

The function call is **double_off()**.

This function is not currently used.

Called By:	none
Routines Called:	printf
Parameters:	none
Returns:	none

2.6.16.6 blank

The blank function clears the screen, starting at a specified location.

The function call is **blank(m)**, where *m* is the starting character position from which the screen is to be blanked.

Called By:	gos_flea_if gos_flea_options	
Routines Called:	printf	
Parameters:	INT_4	m
Returns:	none	

2.6.16.7 save_cur

The save_cur function saves the current cursor position.

The function call is **save_cur()**. This function is not currently used.

Called By:	none
Routines Called:	printf
Parameters:	none

Returns:	none

2.6.16.8 restore_cur

The restore_cur function restores the cursor position to the location saved by save_cur.

The function call is **restore_cur()**. This function is not currently used.

Called By:	none
Routines Called:	printf
Parameters:	none
Returns:	none

2.6.16.9 scroll_reg

The scroll_reg function sets the top and bottom boundaries of the scrolling region.

The function call is **scroll_reg(t, b)**, where:

t is the top of the scroll region
b is the bottom of the scroll region

This function is not currently used.

Called By:	none	
Routines Called:	printf	
Parameters:	INT_4	t
	INT_4	b
Returns:	none	

2.7 Stand-Alone Host Emulator (FLEA) CSC

Flea is an embedded, stand-alone, Simulation Host emulator that resides within the CIG real-time software. Flea permits a system to execute specific features and test limited functionality.

Flea is available only in stand-alone operation mode (i.e., when the system is not being driven through simulation). This mode allows the CIG to generate visual images without interacting with a Simulation Host computer.

Flea is accessed through Gossip, as follows:

1. The user selects the "vehicle demo and fly options" from the Gossip menu.
2. gossip calls gos_fly.
3. The user selects "enter FLEA mode" from the Flying and Demo Selection menu.
4. gos_fly calls gos_flea_if.
5. gos_flea_if asks the user for the vehicle's current location and orientation, then posts a mailbox message to "wake up" flea.

All user commands are entered through Gossip menus. (Refer to sections 2.6.8 and 2.6.9 for details on the Flea user interface.) Flea mode, which requires a VT100-compatible terminal, allows movement around the database via keyboard control.

Flea is not available for Butterfly-based systems.

Figure 2-17 identifies the CSUs in the Flea CSC. These CSUs are described in this section.

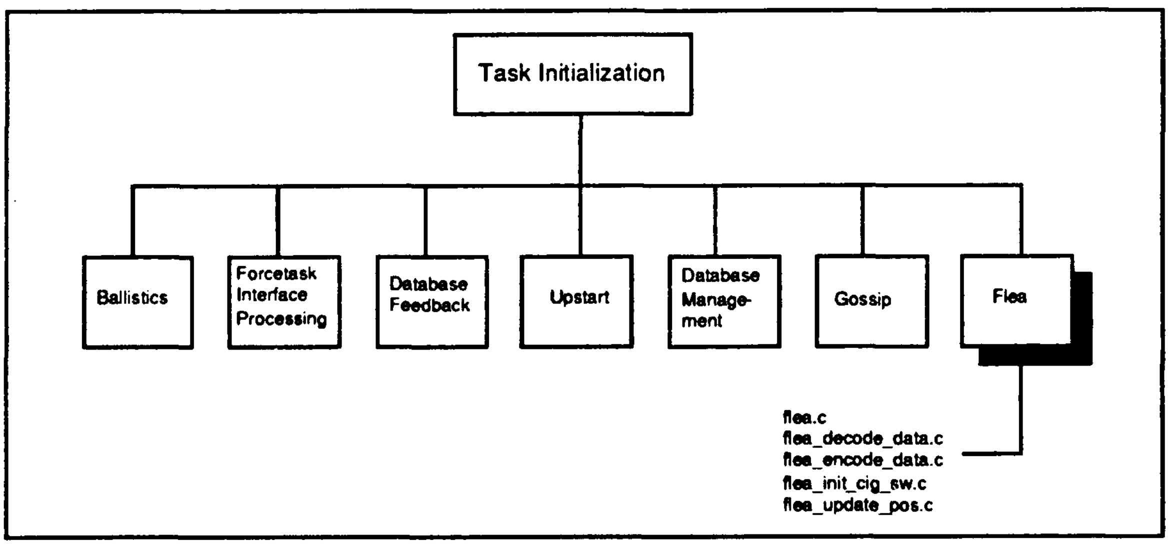


Figure 2-17. Flea CSUs

Figure 2-18 illustrates how the CSUs in the Flea CSC interact.

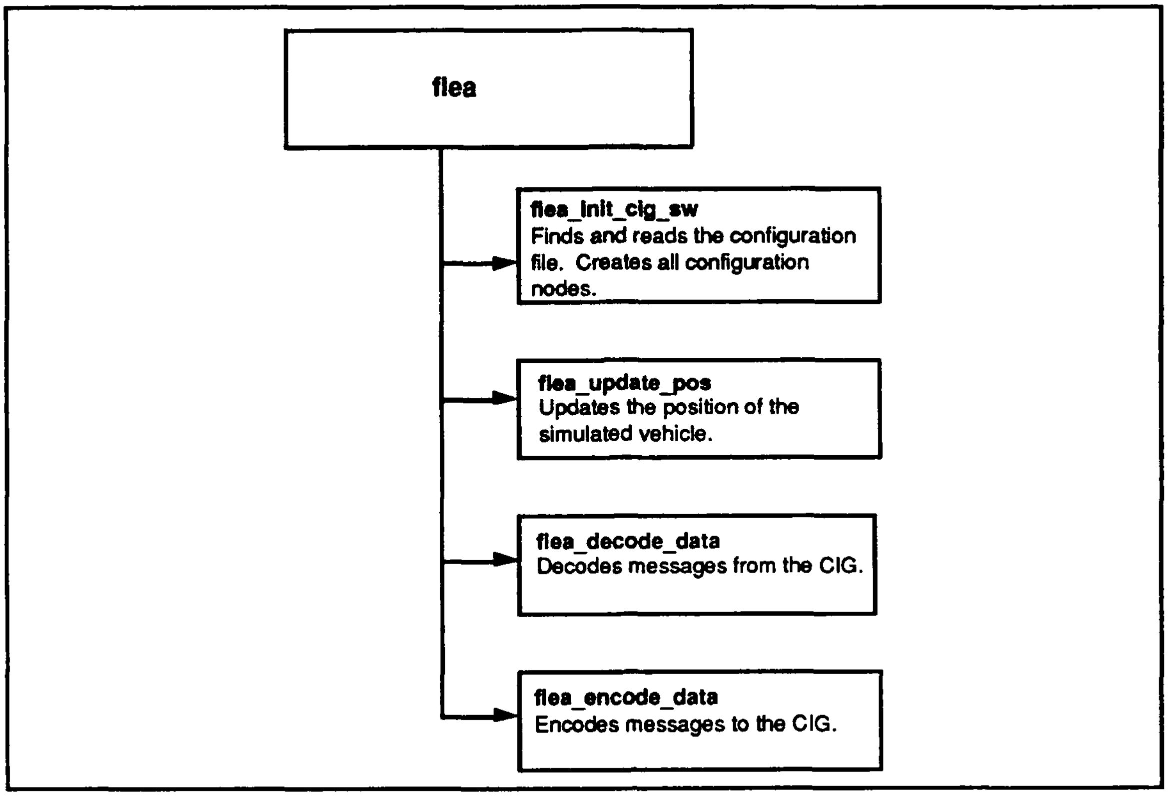


Figure 2-18. Flea Flow Diagram

2.7.1 flea.c

The flea function is a task that runs on the back of the real-time software. It emulates the Simulation Host for stand-alone CIG operation.

The function call is **flea()**. The flea task is created by rtt during the task initialization stage. flea initializes various flags and variables, then suspends itself until gos_flea_if or gos_fly (in the Gossip CSC) posts a FLEA_MB message.

When a FLEA_MB message is posted, flea does the following:

- Calls OPEN_FLEA_DATA to establish the CIG-Flea communications path.
- Calls flea_init_cig_sw to find and read the viewport configuration file.
- Calls EXCHANGE_FLEA_DATA to exchange a message packet with the CIG.
- Calls flea_update_pos to update the position of the simulated vehicle.
- Calls flea_decode_data to decode CIG-to-Flea messages.
- Calls flea_encode_data to encode Flea-to-CIG messages.
- Calls EXCHANGE_FLEA_DATA to exchange a message packet with the CIG.

flea continues to process messages until the system is reset.

Called By: none

Routines Called: EXCHANGE_FLEA_DATA
flea_decode_data
flea_encode_data
flea_init_cig_sw
flea_update_pos
OPEN_FLEA_DATA
sc_pend

Parameters: none

Returns: none

2.7.2 flea_decode_data.c

The flea_decode_data function decodes runtime messages returned from the CIG real-time software. These messages emulate those that would normally be sent to the Simulation Host.

The function call is **flea_decode_data()**. flea_decode_data decodes messages that do the following:

- Report the simulated vehicle's altitude above ground level (MSG_AGL).
- Report a hit (MSG_HIT, MSG_HIT_RETURN, MSG_SHOW_EFFECT).
- Report a miss (MSG_MISS).
- Report on a laser (MSG_LASER_RETURN).
- Describe the local terrain (MSG_LOCAL_TERRAIN, MSG_LT_PIECE).

Called By: flea

Routines Called: none

Parameters: none

Returns: none

2.7.3 flea_encode_data.c

The flea_encode_data function encodes messages to send to the CIG real-time software. These messages emulate runtime messages that would normally be sent by the Simulation Host.

The function call is **flea_encode_data()**. flea_encode_data encodes messages to do the following:

- Update the matrix for the simulated vehicle (MSG_RTS4x3_MATRIX).

- Update the system view flags (MSG_VIEW_FLAGS).
- Process a round (MSG_PROCESS_ROUND).
- Fire a round (MSG_ROUND_FIRED).
- Update the system view mode (MSG_VIEW_MODE).
- Turn on AGL processing (MSG_AGL_SETUP).
- Handle auto-firing (MSG_PROCESS_ROUND).
- Update dynamic vehicle matrices (MSG_OTHERVEH_STATE).
- Add static vehicles (MSG_STATICVEH_STATE).
- Remove static vehicles (MSG_STATICVEH_REM).
- Show effects (MSG_SHOW_EFFECT).
- Display gun overlays (MSG_GUN_OVERLAY).
- Define the ammunition map (MSG_AMMO_DEFINE).

This function also counts hits and misses per minutes.

Called By:	flea
Routines Called:	BCOPY cos sin
Parameters:	none
Returns:	none

2.7.4 flea_init_cig_sw.c

The flea_init_cig_sw function encodes viewport configuration messages.

The function call is **flea_init_cig_sw()**. The function does the following:

- Opens the viewport configuration file.
- Rewinds the file.
- Reads the file size.
- Encodes the configuration messages in the file (MSG_CREATE_CONFIGNODE, MSG_VIEWPORT_STATE, MSG_OVERLAY_SETUP, and MSG_AMMO_DEFINE).

The function returns 1 if successful, or -1 if no configuration file was found.

Called By:	flea
Routines Called:	close cos EXCHANGE_FLEA_DATA find_fn id_4x3mtx

lseek
open
printf
read
rotate_x_nt
rotate_y_nt
rotate_z_nt
sc_pend
sc_post
sin
strlen
translate

Parameters: none

Returns: -1
1

2.7.5 flea_update_pos.c

The flea_update_pos function updates the 4x3 matrix information that is sent each frame to update the position of the simulated vehicle. flea_update_pos also stores the simulated vehicle's current position and orientation if a script is stopped, and restores the simulated vehicle's position and orientation if a script is restarted.

The function call is **flea_update_pos()**.

Called By: flea

Routines Called: cos
id_4x3mtx
rotate_x_nt
rotate_y_nt
rotate_z_nt
sin
translate

Parameters: none

Returns: none

2.8 Force Processor (FORCE) CSC [120TX systems only]

The Force CSC gives the 120TX CIG the ability to display two-dimensional, non-perspective visual data as an overlay on the usual three-dimensional, perspective image. The forcetask is the task that runs on the Force board and serves as the data processing interface between the CIG real-time task and the 2-D processor task. The Force board is the physical interface between the VME chassis and the 2-D processor board.

The real-time software provides 2-D overlay information to the Force board via the forcetask. The forcetask then writes the data to the GSP, the graphics processor chip on the MPV (Micro Processor Video) board. The GSP contains memory for code storage and for storing and manipulating the 2-D image. The Force board can also read data from GSP memory about particular attributes of the displayed image.

The Force and GSP tasks are initially loaded and started by the gsp_load function in the Real-Time Processing component. gsp_load is called by db_mcc_setup before beginning either viewport configuration or 2-D overlay processing, if a Force board is present and GSP has not yet been initialized.

The real-time software communicates with the forcetask via the Force interface structure (defined in mbx.h). The Force front-end control register (FE_CONTROL) is used to specify the command to be performed (SUBSYS_READ_HDATA, SUBSYS_NMI_START, SUBSYS_TEST_MEM, etc.).

Force-GSP processing can also be invoked via the gos_120tx function in Gossip. This function is called when the Gossip user selects the "120tx/t menu" option from the Gossip main menu. The user can then select the "Talk to 2D process/mem" option to display the FORCE-2D Communications Menu. This menu is used to interface with the forcetask.

The forcetask communicates with the GSP to do the following:

Display the 2-D overlays.
The original 2-D overlay configuration is passed to Force by the linkup function in the 2-D Overlay Compiler component. The configuration includes the component pointer table, component descriptor table, and window descriptor table. These structures are downloaded into GSP memory and used to generate the overlays displayed on the viewports.

Change the 2-D overlays during runtime.
Each frame, runtime changes to 2-D components are passed to Force from the real-time software Each message consists of the command (CHANGE_DRAW_2D, DRAW_TEXT_2D, ROTATE_TRANSLATE_2D, etc.) and any arguments (theta, x translation, y translation, etc.) required for that command. Processing for these messages is as follows:

1. The Simulation Host sends a MSG_PASS_ON message to specify the 2-D component changes.
2. The real-time software writes the message to Force memory.
3. The forcetask writes the message to GSP memory.

4. The GSP parses each command in the message, updates the component descriptor table in its memory, then regenerates the 2-D overlays.

A new PASS_ON message is expected every frame. If none is sent, the forcetask reprocesses the last PASS_ON message it received.

The format for each 2-D runtime command is described in the "2-D Commands and Parameters" document.

Return messages to simulation.
Messages such as error reports can be returned from Force to the Simulation Host. The forcetask places the data in Force board memory. The real-time software puts the data into a MSG_PASS_BACK message and returns it to the Simulation Host.

Process laser range request messages.
The Simulation Host can use the MSG_REQUEST_LASER_RANGE message to request the depth of the pixel located at the screen position represented by i, j, where i is the horizontal coordinate (column) and j is the vertical coordinate (row). The real-time software uses the Force interface to request the pixel depth information from the MPV. The real-time software takes the returned value and sends a MSG_LASER_RETURN message to the Simulation Host.

Process mail.
This process triggers the Force/MPV interface to send and receive data such as pass on, pass back, and laser range request messages.

Change the color lookup table.
The MPV's sky color lookup table (LUT) defines the range of 3-D pixel color for each pixel. Available lookup tables are:

OTW	Out the Window
DTV	Daylight TV
WHT	White Hot
BHT	Black Hot

The active lookup table can be changed using the MSG_VIEW_FLAGS message. This message is processed by process_vflags in the Viewport Configuration component of the UPSTART CSC. process_vflags sets the lookup table in Force memory if a Force board is present.

Change the video control registers.
The video control registers can be changed using the MSG_VIEW_FLAGS message. This message is processed by process_vflags in the Viewport Configuration component of the UPSTART CSC. process_vflags sets the video control registers in Force memory if a Force board is present.

Start or stop the GSP task.
gsp_load starts the GSP task initially, and stops and restarts it when testing GSP memory. GSP can also be stopped and restarted via Gossip.

Test reading from/writing to GSP memory.
GSP memory testing is performed by gsp_load at GSP initialization time. Memory testing can also be invoked through Gossip.

Figure 2-19 identifies the CSUs that make up the Force CSC. These CSUs are described in this section.

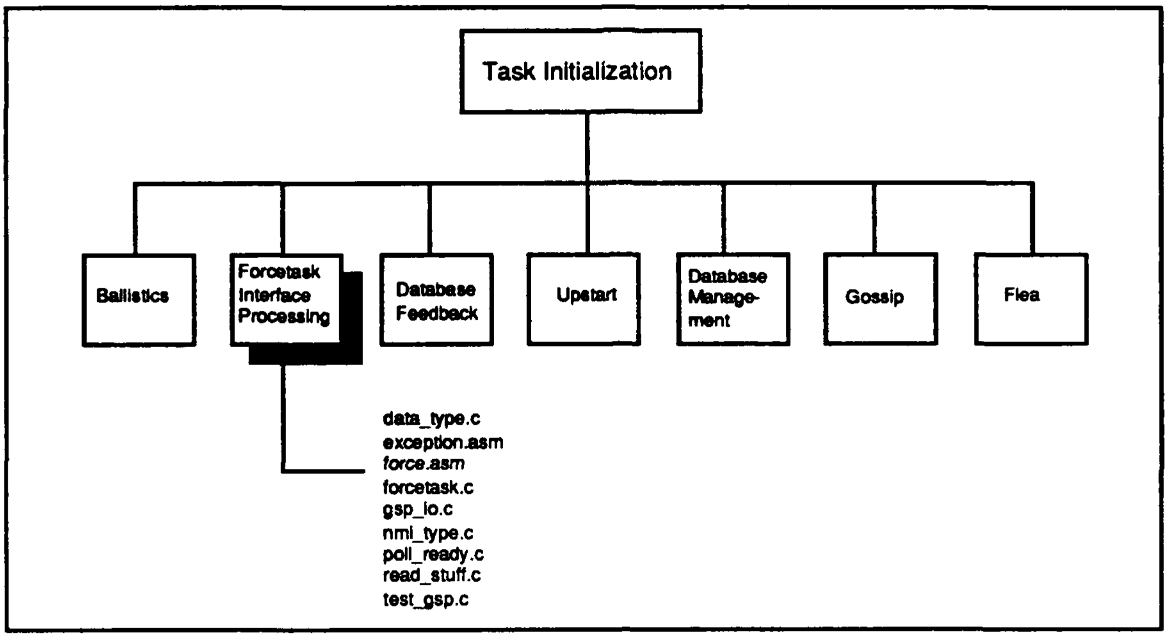


Figure 2-19. Force Processing CSUs

2.8.1 data_type.c

The data_type function reads data from and writes data to GSP memory.

The function call is **data_type()**. data_type does the following:

- Retrieves the type of front-end command: read data or write data.
- Sets the host control value based on whether or not the GSP task is executing, and whether the command is read or write.
- Calls gsp_read or gsp_write to read or write the data as specified by the command.

Called By:	main (in forcetask)
Routines Called:	gsp_ioctl_write gsp_read gsp_write
Parameters:	none
Returns:	none

2.8.2 exception.asm

The exception.asm CSU contains two functions:

- excep_init
- spur_int

2.8.2.1 excep_init

The excep_init function initializes the vector base register (VBR) of the 68010 and all entries of the exception vector table to point to spur_int.

Called By:	main (in forcetask)
Routines Called:	spur_int
Parameters:	none
Returns:	none

2.8.2.2 spur_int

The spur_int function saves all of the 68010 data registers into the structure "context." The order of the save is as follows: D0-D7, A0-A6, SSP, USP, PC, SR.

Called By:	excep_init
Routines Called:	none
Parameters:	none
Returns:	none

2.8.3 force.asm

The force.asm CSU contains a group of subroutines used by the Force functions to read from and write to the GSP. These functions are the following:

- gsp_write
- gsp_read
- gsp_ioctl_write

- gsp_ioctl_read
- init_ports

This module is written in assembly language to obtain the optimal performance from the 68230-to-GSP interface.

2.8.3.1 gsp_write

The gsp_write function writes a block of data from the Force board memory down to the GSP.

The function call is **gsp_write(number_hwords, data_buffer, gsp_address)**, where:

> *number_hwords* is the number of words to be written to the GSP
> *data_buffer* is the location of the data in Force memory
> *gsp_address* is the address to write to

Called By:	data_type main (in forcetask) nmi_type poll_ready test_gsp	
Routines Called:	none	
Parameters:	HWORD HWORD WORD	number_hwords *data_buffer gsp_address
Returns:	none	

2.8.3.2 gsp_read

The gsp_read function reads a block of data from the GSP into Force memory.

The function call is **gsp_read (number_hwords, data_buffer, gsp_address)**, where:

> *number_hwords* is the number of words to be read from the GSP
> *data_buffer* is the location of the data in Force memory
> *gsp_address* is the address to read from

Called By:	data_type main (in forcetask) read_stuff test_gsp

Routines Called:	none	
Parameters:	HWORD HWORD WORD	number_hwords *data_buffer gsp_address
Returns:	none	

2.8.3.3 gsp_ioctl_write

The gsp_ioctl_write function writes the control word to the GSP host interface control register.

The function call is **gsp_ioctl_write(control_data)**, where *control_data* is the control word to be written.

Called By:	data_type gsp_io main (in forcetask) nmi_type poll_ready read_stuff test_gsp	
Routines Called:	none	
Parameters:	int	control_data
Returns:	none	

2.8.3.4 gsp_ioctl_read

The gsp_ioctl_read function reads the control word from the GSP host interface control register. The function returns the control word as an integer (half word = 16 bits).

The function call is **gsp_ioctl_read()**.

Called By:	gsp_io main (in forcetask)
Routines Called:	none

Parameters: none

Returns: control_data

2.8.3.5 init_ports

The init_ports function is called at start-up to initialize the Force-GSP interface.

The function call is **init_ports()**.

Called By: main (in forcetask)

Routines Called: none

Parameters: none

Returns: none

2.8.4 forcetask.c

The forcetask.c CSU contains the main Force program. The two functions in forcetask.c are:

- main
- compare_buffers

2.8.4.1 main

The main function processes commands received from the 2-D overlay compiler or Gossip.

The function call is **main()**. main does the following:

- Sets up registers and initializes various parameters.
- Calls init_ports to initialize the Force-GSP interface.
- Turns off the Force lights.
- Checks the error count.
- Calls gsp_read to check for an illegal opcode trap.
- Calls poll_ready to read the command in the FE_CONTROL register.
- Processes each message, calling other Force functions as appropriate.
- Clears the ready bit.

The following table describes the processing performed by main for each command sent by linkup or gos_120tx. The first column identifies the command, preceded by the value returned by poll_ready (the upper byte of the value in the FE_CONTROL register). The second column describes the purpose of the command (in italics), then shows the steps performed by main to process that command.

Message	Processing by main
0 SUBSYS_MAIL_SEND	*Process mail, pixel depth information, and pass on 2-D components to/from the GSP.* Calls gsp_io.
1 SUBSYS_READ_START, SUBSYS_WRITE_START, SUBSYS_READ_MORE, SUBSYS_WRITE_MORE	*Send data to or receive data from the GSP.* Calls data_type.
2 SUBSYS_NMI_START	*Start the GSP task.* Calls nmi_type.
3 SUBSYS_TEST_MEM	*Test the ability to read from/write to GSP memory.* Calls test_gsp.
6 SUBSYS_STOP	*Halt the GSP task.* Calls gsp_ioctl_write; sets nmi_set_flag to 0.
10 SUBSYS_READ_HCTRL	*Read the control register.* Calls gsp_ioctl_read.
11 SUBSYS_WRITE_HCTRL	*Write to the control register.* Calls gsp_ioctl_write.
12 SUBSYS_READ_HDATA	*Read data from GSP memory.* Calls gsp_read.
13 SUBSYS_WRITE_HDATA	*Write data to GSP memory.* Calls gsp_write; if the verify flag is on, calls gsp_read and compare_buffers to verify the data was written correctly.

Called By: none (the forcetask is loaded and started by gsp_load)

Routines Called:
compare_buffers
data_type
excep_init
gsp_io
gsp_ioctl_read
gsp_ioctl_write
gsp_read
gsp_write
init_ports
nmi_type
poll_ready
test_gsp

Parameters: none

Returns: none

2.8.4.2 compare_buffers

The compare_buffers function is a boolean function that compares the contents of two buffers.

The function call is **compare_buffers(hword_count, ptr1, ptr2)**, where:

hword_count is the length of the data to be compared
ptr1 and *ptr2* are pointers to the buffers to be compared

The function returns 1 if the buffer contents are equal, and 0 if they are not.

Called By:	main (in forcetask)	
Routines Called:	none	
Parameters:	HWORD HWORD HWORD	hword_count *ptr1 *ptr2
Returns:	1 (TRUE) 0 (FALSE)	

2.8.5 gsp_io.c

The gsp_io function processes mail and pixel depth data to and from the GSP.

The function call is **gsp_io()**. gsp_io does the following:

- Sets the data strobe bit to signal the GSP of input/output.
- Gets the buffer id and base address.
- Calls send_stuff to write pixel request data and mail to the GSP.
- Calls read_stuff to read pixel depth data and mail from the GSP.
- Clears the ready bit.

Called By:	main (in forcetask)
Routines Called:	gsp_ioctl_read gsp_ioctl_write read_stuff send_stuff
Parameters:	none

Returns: none

2.8.6 nmi_type.c

The nmi_type function starts the GSP task.

The function call is **nmi_type()**. nmi_type does the following:

- Puts the GSP start address into a data buffer.
- Writes the start address into the nmi vector area of GSP memory.
- Writes to the GSP host interface control register to flush and clear the GSP cache.
- Writes to the GSP host interface control register to start the GSP task.
- Sets the NMI flag for other routines to check before writing to the control register.
- Clears the ready bit.

The NMI (non-maskable interrupt) is a bit in the GSP host interface control register.

Called By: main (in forcetask)

Routines Called: gsp_ioctl_write
gsp_write

Parameters: none

Returns: none

2.8.7 poll_ready.c

The poll_ready function polls the ready bit in the FE_CONTROL register until the bit is set. This register is used to pass messages from the real-time software to the forcetask.

The function call is **poll_ready()**. poll_ready does the following:

- Sets up the address for the FE_CONTROL register.
- While waiting for the ready bit to be set, performs various background functions:
  - Checks for host input regarding color lookup tables, and loads a new table if required.
  - Checks for host input regarding video control registers, and transfers the appropriate values to the MPV (Micro Processor Video) board.
- When it detects that the ready bit is set, returns the upper byte of the control register to the forcetask. This value tells the forcetask what command to process.

Called By: main (in forcetask)

Routines Called: gsp_ioctl_write

gsp_write

Parameters: none

Returns: <upper byte of front-end control register>

2.8.8 read_stuff.c

The read_stuff function is called by gsp_io to read pixel depth data and mail from GSP memory.

The function call is **read_stuff()**. read_stuff does the following:

- Sets the control word for data read.
- Reads the 2D-to-SIM buffer from GSP memory.
- Sets the control word for data read.
- Reads pixel i and pixel j depth from GSP memory.

Called By: gsp_io

Routines Called: gsp_ioctl_write
gsp_read

Parameters: none

Returns: none

2.8.9 test_gsp.c

The test_gsp function writes a pattern to GSP memory, reads it back, and compares values.

The function call is **test_gsp()**. test_gsp does the following:

- Writes a test pattern to a buffer area.
- Sets the host control register for data write.
- Writes the buffer to GSP memory.
- Sets the host control register for data read.
- Reads GSP memory into a second buffer.
- Compares the two buffers and reports the number of errors detected.

Called By: main (in forcetask)

Routines Called: gsp_ioctl_write
gsp_read

gsp_write

Parameters: none

Returns: err_count

3 RESOURCE UTILIZATION

This section summarizes the disk space and memory requirements of the CIG Real-Time software.

3.1 Disk Space Requirements

The total amount of disk space required to house the object files for all of the CIG real-time functions on a 120TX system is approximately 1,593,796 bytes (approximately 1.52 megabytes). On a 120T system, this total is approximately 1,530,170 bytes (1.46 megabytes).

The amount of disk space required to house the terrain database, the dynamic elements database, and the other data files required for a simulation is application-dependent.

3.2 Memory Requirements

The system's memory requirements vary based on application-specific parameters and system options. In general, a minimum of 1 megabyte of CPU memory is required. A minimum of 1.5 megabytes of memory is required for active area memory; additional AAM memory is required for databases with an extended viewing range (greater than 4000 meters).

APPENDIX A. SYSTEM INCLUDE FILES

Include files define data structures and parameters used throughout the system. Although many include files are used exclusively by functions in one area, others are used by multiple CSCs. For easy reference, all of the include files are described in this appendix, in alphabetical order.

A.1 ballistics.h

The ballistics.h file includes all of the common Ballistics header files:

- bx_defines.h
- bx_messages.h
- bx_rtdb_structs.h
- bx_structs.h
- bx_macros.h
- bm_functions.h
- mx_defines.h
- slave133_functions.h (if running on a slave board)

Included By:	All Ballistics Interface Message Processing CSUs All Ballistics Intersection Calculation CSUs bx_init.c bx_task.c gos_bal_query.c

A.2 bbnctype.h

The bbnctype.h file defines character-testing macros (isalpha, isdigit, isascii, etc.) and character-conversion macros (tolower, toupper,and toascii).

Included By:	bbnctype.c read_configfile.c

A.3 bflydisk.h

The bflydisk.h file contains declarations for the Butterfly disk (maximum number of files in a directory and maximum file name size) and provides the typedef for the root directory structure. This file is used for Butterfly Simulation Hosts only.

Included By:	find_fn.c support.c

A.4 bm_functions.h

The bm_functions.h file declares all Ballistics messages (b0_bal_config, b0_database_info, b0_add_traj_table, etc.).

Included By:	ballistics.h bx_init.c

bx_reset.c

A.5 bp_functions.h

The bp_functions.h file is not used by the 120TX/T CIG.

A.6 bx_defines.h

The bx_defines.h file defines the following:

- The MALLOC macro (described in Appendix B).
- The maximum number of bvol types, model types, AAM partitions, messages, static vehicles, rounds, bvol cache entries, poly cache entries, load modules, vehicle load modules, and trajectories.
- DTP data transformation commands.
- DTP data components commands.
- DTP data traversal commands.
- Database effect model numbers.

Included By: ballistics.h

A.7 bx_externs.h

The bx_externs.h file declares external variables for Ballistics, including:

- Input and output buffers.
- Global (G_*) variables.
- Temporary variables used for message processing.

Included By:
All Ballistics Mainline CSUs
All Ballistics Interface Message Processing CSUs
bx_chord_intersect.c
bx_functions.c
bx_get_lm_data.c
bx_model_int.c
bx_reset.c
bx_trajectory.c
gos_bal_query.c

A.8 bx_globals.h

The bx_globals.h file declares variables for Ballistics, including:

- Input and output buffers.
- Global (G_*) variables.
- Temporary variables used for message processing.

Included By: bx_task

A.9 bx_macros.h

The bx_macros.h file defines the following macros used by various functions in Ballistics:

- DELETE_ROUND
- DELETE_STAT_VEH
- FREE_LM_CACHE
- GET_CHORD_END
- GET_DB_POS
- GET_LB_FROM_LM
- NEW_ROUND
- NEW_STAT_VEH

These macros are described in Appendix B.

Included By: ballistics.h

A.10 bx_messages.h

The bx_messages.h file contains the following:

- Declaration of the maximum message size.
- Definitions for the *bal_board_type* (Ballistics board type) variable.
- Definitions for code trace bits.
- The addresses where the boards are locatea.
- Typedefs for all simulation-to-Ballistics (MSG_B0_*) messages.
- Typedefs for all Ballistics-to-simulation (MSG_B1_*) messages.

Included By: bal_routines.c
ballistics.h
db_mcc_setup.c
download_bvols.c
gossip.c
load_modules.c
open_dbase.c
open_ded.c
rowcol_rd.c
simulation.c
upstart.c

A.11 bx_rtdb_structs.h

The bx_rtdb_structs.h file defines the structure of the real-time database for Ballistics. It includes typedefs for the following:

- Floating bounding volume entry.
- Single-transform model structure.
- Show effects stamp structure.
- Tank structure.
- Database directory entry.

- Runtime database header.
- Fixed bounding volume entry.
- Generic load module directory entry.
- Grid components.
- Grid locator information.
- Load module header.
- Load module statistics (generic model, unique static, and terrain grid polygon count, plus total bytes per load module).
- Polygon data (info word).
- Polygon list of vertices and alpha betas for texturing.

This file also defines the maximum number of models that can be put in the generic module of the runtime database, the maximum number of stamps possible in one unique static object definition, and the number of z values in a grid component.

Included By: ballistics.h

A.12 bx_structs.h

The bx_structs.h file contains structure definitions for Ballistics. It includes typedefs for the following:

- Load module/grid search list.
- Static vehicle.
- bvol cache entry.
- Terrain and object polygon.
- Polygon cache entry.
- Load module cache entry.
- Trajectory table entry.
- Trajectory table.
- Point data.
- Chord.
- Round data.
- Terrain corners.

Included By: ballistics.h

A.13 ci_bfly.h

The ci_bfly.h file defines the DGI-Labs message interface. It includes the typedefs for DGI-to-Labs and Labs-to-DGI messages, and defines the mailboxes. This file is required for Butterfly Simulation Hosts only.

Included By: real_time.h

A.14 configtree_def.h

The configtree_def.h file provides definitions used when manipulating the configuration tree, such as matrix and node type values.. It also defines the maximum number of configuration nodes, viewport entries, and graphics path entries.

Included By: real_time.h

A.15 configtree_str.h

The configtree_str.h file describes the structures used in the configuration tree. It provides typedefs for the following:

- Configuration node.
- Overlay parameters.
- Viewport parameters.
- Graphics path parameters.
- View positions (vppos) array.
- Field-of-view vectors.
- Screen and screen constants.

This file also defines the maximum number of graphics paths.

Included By: real_time.h

A.16 ctype.h

The ctype.h file defines character-testing macros (isalpha, isdigit, isspace, etc.) and character-conversion macros (toupper, tolower, toascii).

Included By: get_thing.c

A.17 ded_id_table.h

This file is not currently used.

A.18 defines_2d.h

The defines_2d.h file contains definitions used by the 2-D compiler, including:

- All 2-D database commands (N_*, A_*, and B_*).
- Return codes (end of file, too many errors, invalid window number, etc.).
- Color, plane, and static/dynamic commands.
- MPV addresses (base component pointers and base program area).
- MPV default screen parameters (e.g., dimensions and pitch).
- MPV space allocation.
- Array sizes (maximum number of component pointers, windows, component descriptions, etc.).
- Maximum compiler errors.

Included By: global_2d.h
globfir_2d.h

A.19 definitions.h

The definitions.h file defines miscellaneous constants and structures used by the real-time software. It includes:

- Various definitions used for by Ballistics to parse bounding volume structures and report hits.
- Definitions of various macros (ABSVAL, SET_OUT_BITS, SET_OUT_M2BITS, XREAD, XOPEN, XCLOSE, XLSEEK, XWRITE, AAREAD). These are described in Appendix B.
- The typedef for the load module/grid search list structure.
- Pointers for messages and other parameters.

Included By: real_time.h

A.20 dgi_stdc.h

The dgi_stdc.h file helps make the code compiler-independent by defining basic data types. For Apollo and CIG standard C implementations, the types are defined as follows:

- short INT_2
- int INT_4
- unsigned char BYTE
- unsigned char BOOLEAN
- unsigned short HWORD
- unsigned int WORD
- float REAL_4
- double REAL_8
- char *STRING

Included By:
bit_blt.c
cig_2d_setup.c
cig_comp_2d.c
cig_link_2d.c
comp.c
data_type.c
draw_line.c
forcetask.c
gsp_io.c
get_thing.c
init_stuff.c
nmi_type.c
oval_rect.c
poll_ready.c
poly.c
proc_cmd.c
read_stuff.c
real_time.h
string.c
sysdefs.h
sysdefs2.h

test_gsp.c
text.c
window.c

A.21 dgi_stdg.h

The dgi_stdg.h file defines various graphics structures. It includes typedefs for the following:

- 2-D, 3-D, and 4-D vertex points.
- 4x3 matrix.
- 2-D and 3-D bounding boxes.
- Red, green, blue.
- Red, green, blue, opaque.
- Hue, saturation, lightness.
- Hue, saturation, lightness, opaque.

Included By: real_time.h

A.22 ecompiler1.h

The ecompiler1.h file contains defines for the DTP command generator, including various DTP addresses, maximum values, and the typedef for the *where_process* structure (used for pre- and post-processing models and effects).

This file includes the real_time.h and ememory_map.h files.

Included By: dtp_compiler.c
dtp_funcs.c
dtp_trav1.c
dtp_trav2.c
load_dbase.c
simulation.c

A.23 ememory_map.h

The ememory_map.h file provides external memory declarations. It includes the following:

- General-use variables, such as My Vehicle id, names of the loaded files, and the database column markers.
- Database format variables.
- Database management variables, such as the number of load modules on a side, load module width, and the total number of load modules.
- Variables for ballistics and flea.
- Timing and control word variables.
- Local terrain and range variables.
- Declarations for the DR11-W interface.
- Declarations to support runtime configurable DR packet sizes.
- Intertask semaphore mailbox declarations.
- Debugging and data gathering variables.
- Variables for Flea's keyboard interface.

- FOGM missile mode global variables.
- Variables used with Force and GSP.
- Single step flags.
- Ballistics flags.
- Helicopter blade rotation variables.
- Butterfly-specific declarations.
- The GLOB (global memory) macro, described in Appendix B.

This file includes the memory_map_defines.h file.

Included By: All Flea CSUs
aa_init.c
aam_manager.c
bal_routines.c
bx_task.c
cal.c
cig_config.c
cig_getm_2d.c
concat_mtx.c
db_mcc_setup.c
debug_initdr.c
ded_model_trace.c
dtp_emu.c
ecompiler1.h
file_control.c
fill_tree.c
generic_lm.c
gos_120tx.c
gos_atp.c
gos_bal_query.c
gos_db_query.c
gos_dr11_query.c
gos_flea_if.c
gos_flea_options.c
gos_fly.c
gos_locate.c
gos_memory.c
gos_model.c
gos_polys.c
gos_system.c
gossip.c
gsp_load.c
gun_overlays.c
hw_test.c
load_modules.c
loc_ter.c
make_bbn.c
mkcal.c
model_mtx.c
open_dbase.c
open_ded.c
process_vflags.c
process_vppos.c
rcfuncs.c

read_configfile.c
rowcol_rd.c
support.c
update_fov.c
update_rez.c
upstart.c
viewport_setup.c

A.24 extern.h

The extern.h file defines external variables for the Butterfly interface.

Included By: real_time.h
simulation.c

A.25 external.h

The external.h file is not currently used.

A.26 force.h.asm

The force.h.asm file defines constants for the Force data link. It sets up the 68230 base register and defines GCR codes, address select codes for GSP registers, and LED bit definitions.

Include By: force.asm

A.27 force_defines.h

The force_defines.h file, which contains Force and GSP definitions, serves as the interface between the real-time software, Force, and the GSP. It includes defines for the following:

- FE_CONTROL (the front-end control register).
- FORCE_CONTROL (the Force control register).
- Force return status and error areas.
- Pixel depth request values.
- Lookup table variables.
- Video control variables.
- The READ_CLOCK, RESTART_CLOCK, and CHECK_CLOCK macros.

Several alternate versions of this file exist : force_defines_C.h, force_defines_D.h, force_defines_E.h, and force_defines_TX.h. The only difference between the files is the base address of the Force board. The applicable version of the file is copied to force_defines.h at system build time.

Included By: poll_ready.c

A.28 force_defines_C.h

The force_defines_C.h file replaces the force_defines.h file if the Force board has a VME base address of 0xC00000.

Included By: see force_defines.h

A.29 force_defines_D.h

The force_defines_D.h file replaces the force_defines.h file if the Force board has a VME base address of 0xD00000.

Included By: see force_defines.h

A.30 force_defines_E.h

The force_defines_E.h file replaces the force_defines.h file if the Force board has a VME base address of 0xE00000.

Included By: see force_defines.h

A.31 force_defines_TX.h

The force_defines_TX.h file replaces the force_defines.h file if the Force board has a VME base address of 0x100000.

Included By: see force_defines.h

A.32 functions.h

The functions.h file defines the following macros used by various functions in the real-time software:

- DART_ENQUEUE
- DUMP_DART_BUFFER
- EXCHANGE_DATA
- EXCHANGE_DATA_SIM
- EXCHANGE_FLEA_DATA
- FIND_LM
- FLTOFX
- FXTO881
- FXTOFL
- INIT_MTX
- OPEN_EXCHANGE
- OPEN_FLEA_DATA
- SYSERR
- TRIGGER_FORCE
- WAIT_FORCE

These macros are described in Appendix B.

Included By: real_time.h

A.33 ghctype.h

The ghctype.h file is not currently used.

A.34 global_2d.h

The global_2d.h file includes the defines_2d.h and struct_2d.h include files. Collectively, these files declare all global I/O variables, global temporary compiler variables, and compiler product variables for the 2-D compiler.

Included By: bit_blt.c
cig_comp_2d.c
cig_getm_2d.c
cig_link_2d.c
comp.c
draw_line.c
get_thing.c
init_stuff.c
oval_rect.c
poly.c
proc_cmd.c
string.c
text.c
window.c

A.35 globfir_2d.h

The globfir_2d.h file includes the defines_2d.h and struct_2d.h include files. Collectively, these files declare all global I/O variables, global temporary compiler variables, and compiler product variables for the 2-D compiler.

Included By: cig_2d_setup.c

A.36 m2_config.h

The m2_config.h file contains defines specific to the M2. It defines channel and gunner resolution, vi^r vport angular offsets, pitch up/down angular offsets, field-of-view sizes for all channels, and texture map definitions.

Included By: gun_overlays.c

A.37 mbx.h

The mbx.h file contains defines used for the Force board. It includes defines for the following:

- The FORCE_INTERFACE structure.
- NMI and GSP configuration addresses.
- Force masks (slave/host, resolution, front ready, force busy, etc.).
- Force commands (SUBSYS_READ_START, SUBSYS_TEST_MEM, etc.).
- Video control parameters (VIDEO_CTL_ADDR, VIDEO_ON_CODE, etc.).
- GSP memory start and end addresses.

Included By: cig_link_2d.c
data_type.c
forcetask.c
gsp_io.c
nmi_type.c
poll_ready.c
real_time.h
read_stuff.c
test_gsp.c.c

A.38 memory_map.h

The memory_map.h file contains external memory declarations. It defines the following:

- Variables describing the simulated vehicle (*myveh_id*, *myveh_type*, etc.).
- Database header and table structure variables.
- Database management variables.
- Variables for Ballistics and Flea.
- Timing and control word variables.
- Local terrain and range variables.
- Declarations for the DR11-W interface.
- Default DR11-W interface packet size.
- Default local terrain chunk size and interval.
- Intertask semaphore mailbox declarations.
- Viewport position, rotation data for flying and setting individual views.
- Variables used by Flea's keyboard interface for flying.
- FOGM missile mode global variables.
- Helicopter blade rotation variables.
- Various Ballistics variables.
- The GLOB (global memory) macro, defined in Appendix B.

Included By: upstart.c

A.39 memory_map_defines.h

The memory_map_defines.h file defines variables used in external memory declarations. It defines the following:

- The default T&C location.
- The size of a load module.
- The areas of the 64KW memory board (32KW of space for the double-buffer state table and 32KW of generic memory for the database).
- Byte offsets to data in double-buffered state table memory.
- Declarations for the DR11-W interface.
- Local terrain message interval and starting frame number.
- Intertask semaphore mailbox locations.
- Viewport position, rotation data for flying and setting individual views.
- Helicopter blade rotation variables, used in simulation.
- Butterfly-specific variables used for the VME interface.
- Ammunition maps for the M2 gunner's overlay (high-explosive 25mm, tow missile, sabot, and coax machine gun).

Included By: ememory_map.h
memory_map.h

A.40 mx_defines.h

The mx_defines.h file defines the following:

- Constants used for Ballistics message queue processing (MX_DEVICE_CLOSED, MX_DEVICE_TABLE_FULL, etc.).
- The MX_DEVICE and MESSAGE_HEADER structures.
- The BCOPY macro, described in Appendix B.

Included By: All Ballistics Message Queue Processing CSUs
bal_routines.c
ballistics.h
download_bvols.c
flea_encode_data.c
gos_flea_options.c
load_modules.c
open_dbase.c
open_ded.c
rowcol_rd.c
simulation.c
upstart.c

A.41 ovrly_defs.h

The ovrly_defs.h file contains definitions used to create calibration overlays (for example, the dimensions of the frame triangles).

Included By: real_time.h

A.42 rcinclude.h

The rcinclude.h file is used by the DTP command generator and the Runtime Command Library. It does the following:

- Declares all RCL functions (rcl_push, rcl_pop, etc.).
- Declares address and pointer variables used by the RCL commands.
- Defines the RCL_UNION structure.
- Defines the macros used by dtp_trav1 and dtp_trav2 to generate RCL commands. These macros, which are defined in Appendix B, are used to pass the appropriate data to rcl_command, rcl_lblcmd, rcl_data, and rcl_component.

Included By: dtp_compiler.c
dtp_funcs.c
dtp_trav1.c
dtp_trav2.c
rcfuncs.c

A.43 real_time.h

The real_time.h file includes many of the include files used in the real-time software. The files it includes are the following:

- ci_bfly.h (for Butterfly compatibility)
- configtree_def.h
- configtree_str.h
- definitions.h
- dgi_stdc.h
- dgi_stdg.h
- extern.h (for Butterfly compatibility)
- functions.h
- mbx.h
- ovrly_defs.h
- rtdb_struct.h
- sim_cig_if.h
- structures.h

Included By: All Ballistics Interface Message Processing CSUs
All Ballistics Message Queue Processing CSUs
All Ballistics Intersection Calculations CSUs
All Gossip CSUs
All Flea CSUs
aa_init.c
aam_manager.c
bal_get_db_pos.c
bal_get_lm_grid.c
bal_routines.c
bx_init.c
bx_task.c
cal.c
cig_config.c
cig_getm_2d.c
concat_mtx.c
confignode_setup.c
db_mcc_setup.c
debug_initdr.c
ded_model_trace.c
download_bvols.c

ecompiler1.h
file_control.c
fill_tree.c
find_fn.c
fxbvtofl.c
generic_lm.c
gsp_load.c
gun_overlays.c
hw_test.c
load_modules.c
loc_ter.c
make_bbn.c
mat_dump.c
mkcal.c
mkmtx_nt.c
model_mtx.c
open_dbase.c
open_ded.c
overlay_setup.c
process_vflags.c
process_vppos.c
rcfuncs.c
read_configfile.c
rowcol_rd.c
slave133_functions.c
support.c
update_fov.c
update_rez.c
upstart.c
vec_dump.c
viewport_setup.c

A.44 rt_definitions.h

The rt_definitions.h file is not used by the 120TX/T CIG software.

A.45 rt_macros.h

The rt_macros.h file is not used by the 120TX/T CIG software.

A.46 rt_types.h

The rt_types.h file is not used by the 120TX/T CIG software.

A.47 rtdb_struct.h

The rtdb_struct.h file defines the following real-time database structures:

- Database version and tag.
- Database header data.

- Database header overflow and landmark data.
- Generic module directory entry data and name.
- Model and catalog tables.
- Database directory entry.
- Load module header.
- Grid locator information.
- Fixed bvol entry
- Load module statistics.
- Floating bvol entry.

This file also defines the maximum number of models that can be put in the generic module of the runtime database, the maximum number of stamps in one unique static object definition, and the number of z values in a grid component.

Included By: real_time.h

A.48 sim_cig_ari.h

The sim_cig_ari.h file is an alternate form of the sim_cig_if.h file, used for a specific customer (Army Research Institute).

Included By: see sim_cig_if.h

A.49 sim_cig_ari_if.h

The sim_cig_ari_if.h file is an alternate form of the sim_cig_if.h file, used for a specific customer (Army Research Institute). This version differs from sim_cig_ari.h only in the definition of the packet buffer size.

Included By: see sim_cig_if.h

A.50 sim_cig_if.h

The sim_cig_if.h file defines the interface between the CIG and the Simulation Host. It defines the following:

- All SIM-to-CIG, CIG-to-SIM, and configuration tree message structures.
- The maximum number of tanks, non-tank vehicles, concurrent active effects, static tanks, and static vehicles.
- Vehicle types (main battle tank, personnel carrier, etc.).
- Vehicle appearance modifiers (destroyed, flaming, dust cloud, etc.).
- Vehicle special modifier codes (small tree, rock, house, etc.).
- Special effects (explosion on ground, fire, smoke plume, etc.).
- Types of ammunition that cause effects (heat105, sabot25, etc.).
- Application-specific data (ASID) types (data unique to a particular model).
- The structures of the matrix formats.

Included By: real_time.h

A.51 sim_cig_if512x512.h

The sim_cig_if512x512.h file is obsolete. It is not used by the 120TX/T CIG software.

A.52 sim_cig_if7kx1k.h

The sim_cig_if7kx1k.h file is obsolete. It is not used by the 120TX/T CIG software.

A.53 slave133_functions.h

The slave133_functions.h file declares the slave133_malloc() function. This file is included by ballistics.h if Ballistics is running on a slave board.

Included By:	ballistics.h

A.54 struct_2d.h

The struct_2d.h file defines the window structures used by the 2-D compiler.

Included By:	global_2d.h globfir_2d.h

A.55 structures.h

The structures.h file defines various data structures used to process overlays and static and dynamic models. It includes typedefs for the following structures:

- Component data type (3-D point, 2-D point, and vector).
- Texture map index.
- Polygon information word.
- Polygon and stamp lists.
- Gunner, bun barrel, and calibration overlays.
- Field-of-view test table.
- Load module call tables.
- Static and dynamic tanks.
- Static and dynamic single-transform models.
- Remove static model.
- Show effects (stamp structure).
- Ballistics chord data.
- Trajectory positions and data.
- Load module-specific data.
- Grid component definition.

This file also defines the following:

- DTP data transformation commands.
- DTP data component commands.
- DTP data traversal commands.

- Ballistics and local terrain data pointers.
- Bounding plane definitions.
- Channel definitions.

Included By: real_time.h

A.56 sysdefs.h

The sysdefs.h file provides system definitions for operating system versions RTOS.101 and RTOS.102. It includes the following:

- System-wide memory, resource, and software and hardware fault definitions.
- Task definitions.
- I/O control system definitions.
- VRTX return codes.
- Disk manager fault codes.
- File control system error codes.
- Special character definitions.
- 68901 equates.
- System interrupt equates.
- Definitions and structures used by file_control.

Included By: rtt.c

A.57 sysdefs2.h

The sysdefs2.h file provides system definitions for operating system version FOS.100, which allows the use of high-speed disks. It includes the following:

- System-wide memory, resource, and software and hardware fault definitions.
- Task definitions.
- I/O control system definitions.
- VRTX return codes.
- Disk manager fault codes.
- File control system error codes.
- Special character definitions.
- 68901 equates.
- System interrupt equates.
- Definitions and structures used by file_control.

Included By: getch.c

A.58 tflat.h

The tflat.h file defines Ballistics round trajectories for a completely flat trajectory. This is a default table loaded for testing purposes.

Included By: bx_init.c

A.59 tflat_slow.h

The tflat_slow.h file defines Ballistics round trajectories for a completely flat trajectory with a very slow fly-out. This is a default table loaded for testing purposes.

Included By: bx_init.c

A.60 u105mmsabot30hz.h

The u105mmsabot30hz.h file defines Ballistics round trajectories for a u105mmsabot round with a 30 Hz sample rate. This is a default table loaded for testing purposes.

Included By: bx_init.c

A.61 u25mmheat.h

The u25mmheat.h file defines Ballistics round trajectories for a u25mmheat round with a 15 Hz sample rate. This is a default table loaded for testing purposes.

Included By: bx_init.c

APPENDIX B. SYSTEM MACROS

Macros are used throughout the system to perform specialized functions. Most macros are defined in one of the following files:

bx_macros.h
Macros used exclusively by Ballistics.

functions.h
Macros used throughout the real-time software.

rcfuncs.c and rcinclude.h
Macros used by the Runtime Command Library and DTP.

Although some macros are used exclusively in one area of the system, others are used by multiple CSCs. For easy reference, all macros are described in this appendix, in alphabetical order.

B.1 AAREAD

The AAREAD macro is defined as the system call "read" for the 120T CIG MVME133, and "fread" for the Butterfly.

Defined In:	definitions.h
Called By:	none
Routines Called:	fread read
Parameters:	none

B.2 ABSVAL

The ABSVAL macro determines the absolute value of a number. The usage is **ABSVAL(x)**, where x is the number.

Defined In:	definitions.h	
Called By:	none	
Routines Called:	none	
Parameters:	int	x

B.3 BCOPY

The BCOPY macro copies a specified number of bytes. The usage is **BCOPY(source, dest, byte_count)**, where:

source is a pointer to the source location
dest is a pointer to the destination location
byte_count is the number of bytes to be copied

Defined In:	mx_defines.h	
Called By:	b0_add_static_vehicle b0_bal_config b0_bvol_entry b0_database_info b0_model_entry bx_chord_intersect download_bvols flea_encode_data mx_push	
Routines Called:	none	
Parameters:	WORD WORD HWORD	*source *dest byte_count

B.4 CHECK_CLOCK

The CHECK_CLOCK macro, defined in force_defines.h, is not currently used.

B.5 CHECK_FORCE

The CHECK_FORCE macro checks to see if the forcetask is running by reading the ready bit (FRONT_RDY_MASK) in the front-end control register (FE_CONTROL). If it is, the Gossip operation is denied and the user is asked to retry later.

The usage is **CHECK_FORCE**.

Defined In:	gos_120tx.c
Called By:	gos_120tx

Routines Called: printf

Parameters: none

B.6 DART_ENQUEUE

The DART_ENQUEUE macro, defined in functions.h, is not currently used. Previously, this macro was used to add a message to the DART Ballistics board's queue. The DART Ballistics board is no longer supported.

B.7 DELETE_ROUND

The DELETE_ROUND macro removes a round from the active list and puts it on the free list. The usage is **DELETE_ROUND(dead_round_P)**, where *dead_round_P* is a pointer to the round to be deleted.

Defined In: bx_macros.h

Called By: b0_new_frame
b0_process_round
b0_round_fired

Routines Called: none

Parameters: ROUND_DATA *dead_round_P

B.8 DELETE_STAT_VEH

The DELETE_STAT_VEH macro removes a static vehicle from a load module list and puts it in the free list. The usage is **DELETE_STAT_VEH(dead_sv_P, table_P)**, where:

dead_sv_P is a pointer to the static vehicle to be deleted
table_P is a pointer to the vehicle table

Defined In: bx_macros.h

Called By: b0_delete_static_vehicle

Routines Called: none

Parameters: STAT_VEH *dead_sv_P
STRUCT_P_SV *table_P

B.9 DOWNLOAD_DATA

The DOWNLOAD_DATA m cro downloads 2-D overlay data into GSP memory. The usage is **DOWNLOAD_DATA.**

Defined In: cig_link_2d.c

Called By: linkup

Routines Called: WAIT_FORCE

Parameters: none

B.10 dtp.* (DTP Macros)

Macros are used by the DTP Command Generator functions to interface to the Runtime Command Library (RCL). The macros call RCL routines to generate the actual commands that are downloaded to the hardware.

Each DTP hardware command has one or more supporting macros. The macro called by the DTP Command Generator functions depends on the desired command, whether a label is being used, and whether relative or absolute addressing is being used.

The following table lists each DTP macro and identifies its parameters, calling routines, and called routines. It also identifies the DTP command generated by RCL for each macro. Detailed descriptions of the hardware commands are beyond the scope of this document.

Defined In: rcinclude.h

Called By: see table below

Routines Called: see table below

Parameters: see table below

Macro(parameters)	DTP Hardware Command Generated	Called By	Routines Called
dtp_bcn(label, mask, channel_data_offset)	Branch Channel Non-Zero	none	rcl_lblcmd
dtp_bcnr(label, mask, channel_data_offset)	Branch Channel Non-Zero Relative	none	rcl_lblcmd
dtp_bcnrs(aam_address, mask, channel_data_offset)	Branch Channel Non-Zero Relative	none	rcl_command
dtp_bcns(aam_address, mask, channel_data_offset)	Branch Channel Non-Zero	none	rcl_command
dtp_bcz(label, mask, channel_data_offset)	Branch Channel Zero	none	rcl_lblcmd
dtp_bczr(label, mask, channel_data_offset)	Branch Channel Zero Relative	none	rcl_lblcmd
dtp_bczrs(aam_address, mask, channel_data_offset)	Branch Channel Zero Relative	none	rcl_command
dtp_bczs(aam_address, mask, channel_data_offset)	Branch Channel Zero	none	rcl_command
dtp_bdgr(label, cos_squared)	Branch DOT Greater Than Relative	none	rcl_lblcmd
dtp_bdgrs(pc_offset, cos_squared)	Branch DOT Greater Than Relative	none	rcl_command
dtp_bdlr(label, cos_squared)	Branch DOT Less Than Relative	none	rcl_lblcmd
dtp_bdlrs(pc_offset, cos_squared)	Branch DOT Less Than Relative	none	rcl_command
dtp_bgn(label, mask)	Branch Generic Non-Zero	none	rcl_lblcmd
dtp_bgns(aam_address, mask)	Branch Generic Non-Zero	none	rcl_command
dtp_bgz(label, mask)	Branch Generic Zero	none	rcl_lblcmd
dtp_bgzs(aam_address, mask)	Branch Generic Zero	none	rcl_command
dtp_blm(dtp_viewpoint_address, dtp_result_address, x_multiplier, y_multiplier)	Base Load Module Calculation	dtp_trav2	rcl_command
dtp_bnz(label, mask, dtp_address)	Branch Non-Zero	dtp_trav1, dtp_trav2	rcl_lblcmd
dtp_bnzr(label, mask, dtp_address)	Branch Non-Zero Relative	none	rcl_lblcmd
dtp_bnzrs(aam_address, mask, dtp_address)	Branch Non-Zero Relative	none	rcl_command
dtp_bnzs(aam_address, mask, dtp_address)	Branch Non-Zero Relative	none	rcl_command
dtp_bpc()	Bounding Plane Normals Calculation	dtp_trav2	rcl_command
dtp_bpcx()	Bounding Plane Normals Calculation TX	none	rcl_command

dtp_brulabel)	Branch Unconditionally	dtp_trav1, dtp_trav2	rcl_lblcmd
dtp_brur(label)	Branch Unconditionally Relative	none	rcl_lblcmd
dtp_brurs(pc_offset)	Branch Unconditionally	dtp_trav2	rcl_command
dtp_brus(aam_address)	Branch Unconditionally Relative	dtp_trav1	rcl_command
dtp_brz(label, mask, dtp_address)	Branch Zero	dtp_trav2	rcl_lblcmd
dtp_brzr(label, mask, dtp_address	Branch Zero Relative	none	rcl_lblcmd
dtp_brzrs(pc_offset, mask, dtp_address)	Branch Zero Relative	none	rcl_command
dtp_brzs(aam_address, mask, dtp_address)	Branch Zero	none	rcl_command
dtp_dot(vx, vy, vz)	Dot Product	none	rcl_command
dtp_elm()	End Load Module	none	rcl_command
dtp_end()	End Current Path	dtp_trav1, dtp_trav2	rcl_command
dtp_fov(label, radius)	Field of View Test	none	rcl_lblcmd
dtp_fovr(label, radius)	Field of View Test Relative	none	rcl_lblcmd
dtp_fovrs(pc_offset, radius)	Field of View Test Relative	none	rcl_command
dtp_fovs(aam_address, radius)	Field of View Test	none	rcl_command
dtp_gdc(label, centroid_x, centroid_y, centroid_z, asid)	Generic Data Call	none	rcl_lblcmd
dtp_gdci(label, centroid_x, centroid_y, centroid_z, asid, dptr)	Generic Data Call	none	rcl_lblcmd
dtp_gdcir(label, centroid_x, centroid_y, centroid_z, asid, dptr)	Generic Data Call Relative	none	rcl_lblcmd
dtp_gdcirs(aam_address, centroid_x, centroid_y, centroid_z, asid, dptr)	Generic Data Call Relative	none	rcl_command
dtp_gdcis(aam_address, centroid_x, centroid_y, centroid_z, asid, dptr)	Generic Data Call	none	rcl_command
dtp_gdcn(label, centroid_x, centroid_y, centroid_z)	Generic Data Call	none	rcl_lblcmd
dtp_gdcnr(label, centroid_x, centroid_y, centroid_z)	Generic Data Call Relative	none	rcl_lblcmd
dtp_gdcnrs(aam_address, centroid_x, centroid_y, centroid_z)	Generic Data Call Relative	none	rcl_command
dtp_gdcns(aam_address, centroid_x, centroid_y, centroid_z)	Generic Data Call	none	rcl_command
dtp_gdcr(label, centroid_x, centroid_y, centroid_z, asid)	Generic Data Call Relative	none	rcl_lblcmd
dtp_gdcrs(aam_address, centroid_x, centroid_y, centroid_z, asid)	Generic Data Call Relative	none	rcl_command
dtp_gdcs(aam_address, centroid_x, centroid_y, centroid_z, asid)	Generic Data Call	none	rcl_command
dtp_gr(offset)	Generic Return	none	rcl_command

dtp_lmi(label, radius)	Load Module In Field of View Test	none	rcl_lblcmd
dtp_lmir(label, radius)	Load Module In Field of View Test Relative	none	rcl_lblcmd
dtp_lmirspc_offset, radius)	Load Module In Field of View Test Relative	none	rcl_command
dtp_lmis(aam_address, radius)	Load Module In Field of View Test	none	rcl_command
dtp_lod(label, range_squared)	Level of Detail Test	none	rcl_lblcmd
dtp_lodr(label, range_squared)	Level of Detail Test Relative	none	rcl_lblcmd
dtp_lodrs(pc_offset, range_squared)	Level of Detail Test Relative	none	rcl_command
dtp_lods(aam_address, range_squared)	Level of Detail Test	none	rcl_command
dtp_lwd(label, dtp_address, word_count)	Load Words	dtp_trav1	rcl_lblcmd
dtp_lwdr(label, dtp_address, word_count)	Load Words Relative	none	rcl_lblcmd
dtp_lwdrs(pc_offset, dtp_address, word_count)	Load Words Relative	none	rcl_command
dtp_lwds(aam_address, dtp_address, word_count)	Load Words	dtp_trav1, dtp_trav2	rcl_command
dtp_mml(dtp_address_a, dtp_address_b, dtp_address_c)	Matrix Multiply Local (A*B=>C)	none	rcl_command
dtp_mmpre(dtp_address_a, dtp_address_b, dtp_address_c)	Matrix Multiply Pre (A*B=>C)	none	rcl_command
dtp_mmpst(dtp_address_a, dtp_address_b, dtp_address_c)	Matrix Multiply Post (A*B=>C)	dtp_trav1, dtp_trav2	rcl_command
dtp_mwd(dtp_address_a, dtp_address_b, word_count)	Move Words	dtp_trav1	rcl_command
dtp_ngc(centroid_x, centroid_y, centroid_z)	Non-Generic Centroid	none	rcl_command
dtp_oio(output_offset, word_count)	Output Indirect Offset	none	rcl_command
dtp_oos(output_offset, word_count, stack_offset)	Output Offset Stack	none	rcl_command
dtp_osd(label)	Output Single Word Direct	dtp_trav2	rcl_lblcmd
dtp_osds(aam_address)	Output Single Word Direct	none	rcl_command
dtp_owd(label, word_count)	Output Words Direct	dtp_trav2	rcl_lblcmd
dtp_owds(aam_address, word_count)	Output Words Direct	dtp_trav2	rcl_command
dtp_owdsc(label, end_label)	Output Words Direct - Set Count	none	rcl_lblcmd
dtp_owo(aam_address_offset, word_count)	Output Words Offset	none	rcl_command
dtp_owr(label, word_count)	Output Words Relative	none	rcl_lblcmd
dtp_owrs(pc_offset, word_count)	Output Words Relative	none	rcl_command
dtp_owrsclabel, end_label)	Output Words Relative - Set Count	none	rcl_lblcmd

dtp_rc()	Range Calculation	none	rcl_command
dtp_sub(label)	Subroutine Call	none	rcl_lblcmd
dtp_subr(label)	Subroutine Call Relative	none	rcl_lblcmd
dtp_subrs(pc_offset)	Subroutine Call Relative	none	rcl_command
dtp_subs(aam_address)	Subroutine Call	dtp_trav2	rcl_command
dtp_tbc(total_time)	Time Base Calculation	none	rcl_command
dtp_tbdr(label, start_time, end_time)	Time Base Data Relative	none	rcl_lblcmd
dtp_tbdrs(pc_offset, start_time, end_time)	Time Base Data Relative	none	rcl_command
dtp_tbrr(label, maximum_time)	Time Branch Relative	none	rcl_lblcmd
dtp_tbrrs(pc_offset, maximum_time)	Time Branch Relative	none	rcl_command

B.11 DUMP_DART_BUFFER

The DUMP_DART_BUFFER macro, defined in functions.h, is not currently used. Previously, this macro was used for DART Ballistics boards, which are no longer supported.

B.12 ERRMSG

The ERRMSG macro prints an error for the DTP/RCL functions. The usage is **ERRMSG(a, b)**, where:

a is the error message text
b is the name of the calling routine

Defined In: rcfuncs.c

Called By: rcl_patch_adrs
rcl_pop
rcl_push
rcl_set_cntlbl
rcl_set_label

Routines Called: printf

Parameters: char a[]
char b[]

B.13 EXCHANGE_DATA

The EXCHANGE_DATA macro is used to exchange message packets with the Simulation Host. It loads the end message to the output buffer and sends it , then obtains an input message packet.

The usage is **EXCHANGE_DATA(state)**, where *state* is the current state of the CIG.

Defined In:	functions.h
Called By:	get_msg_2d cig_config db_mcc_setup file_control hw_test upstart
Routines Called:	debug_initdr printf read sc_pend sc_post SYSERR write
Parameters:	INT_2 state

B.14 EXCHANGE_DATA_SIM

The EXCHANGE_DATA_SIM macro is used by simulation to exchange message packets with the Simulation Host. It loads the end message to the output buffer and sends it , then obtains an input message packet. It also determines if it is time to send a local terrain message.

The usage is **EXCHANGE_DATA_SIM(state)**, where *state* is the current state of the CIG.

Defined In:	functions.h
Called By:	simulation
Routines Called:	printf read sc_pend sc_post SYSERR write
Parameters:	INT_2 state

B.15 EXCHANGE_FLEA_DATA

The EXCHANGE_FLEA_DATA macro is used by Flea to exchange message packets with the CIG. It loads the end message to the output buffer and sends it , then obtains an input message packet.

The usage is **EXCHANGE_FLEA_DATA(flea_imsg, flea_omsg)**, where:

flea_imsg is a pointer to the input message packet
flea_omsg is a pointer to the output message packet

Defined In:	functions.h	
Called By:	flea flea_init_cig_sw	
Routines Called:	sc_pend sc_post	
Parameters:	INT_4 INT_4	*flea_imsg *flea_omsg

B.16 FIND_LM

The FIND_LM macro finds the load module in which a given x, y location lies. It is assumed that the point is within active area memory.

The usage is **FIND_LM (x, y, lm, inv_width, mask, num_per_side)**, where:

x is the location's x coordinate
y is the location's y coordinate
lm is the number of the load module
inv_width is the inverse of the width of a load module
mask is the mask of the number of load module blocks per side (currently always 0x0F)
num_per_side is the number of load modules per side of AAM

Defined In:	functions.h
Called By:	bal_get_db_pos bx_get_db_pos gos_bal_query simulation

Routines Called:	none	
Parameters:	INT_4	x
	INT_4	y
	INT_4	lm
	REAL_4	inv_width
	INT_4	mask
	HWORD	num_per_side

B.17 FLTOFX

The FLTOFX macro, defined in functions.h, is no longer used. Previously, this macro was used to convert a floating point value to fixed point. The FXTO881 macro is now used to perform this operation.

B.18 FREE_LM_CACHE

The FREE_LM_CACHE macro, when given a load module in the Ballistics database cache, puts the bounding volumes in that module on the free bvol list, and puts the polygons in that module on the free polygon lists.

The usage is **FREE_LM_CACHE(lm_dir)**, where *lm_dir* is a load module in the cache.

Defined In:	bx_macros.h	
Called By:	b0_lm_read	
	bx_new_bvol	
	bx_new_poly	
Routines Called:	none	
Parameters:	LM_CACHE_ENTRY	*lm_dir

B.19 FXTO881

The FXTO881 macro converts a fixed point value to floating point. The usage is **FXTO881(fxd, flt, bits)**, where:

fxd is the fixed point value to be converted
flt is the floating point value (result)
bits is the number of fractional bits in the fixed point number

Defined In:	functions.h

Called By:	bx_get_lm_data fxbvtofl fxbvtofl_020 fxbvtofl_dart local_terrain	
Routines Called:	none	
Parameters:	INT_2 REAL_4 INT_4	fxd flt bits

B.20 FXTOFL

FXTOFL converts a fixed point value to floating point. The usage is **FXTOFL(fxd, flt, nfract_bits, exp, tmp)**, where:

fxd is the fixed point value to be converted
flt is the floating point value (result)
nfract_bits is the number of fractional bits in the fixed point number
exp is a temporary variable used for calculations
tmp is a temporary variable used for calculations

Defined In:	functions.h	
Called By:	local_terrain simulation	
Routines Called:	none	
Parameters:	INT_4 REAL_4 INT_2 INT_4 INT_4	fxd flt nfract_bits exp tmp

B.21 GET_CHORD_END

The GET_CHORD_END macro finds the next chord in the trajectory. The usage is **GET_CHORD_END(chord_P)**, where *chord_P* is a pointer to the chord.

This macro is not currently used.

Defined In:	bx_macros.h

Called By:	none	
Routines Called:	none	
Parameters:	CHORD	*chord_P

B.22 GET_DB_POS

The GET_DB_POS macro finds the load module that corresponds to a given point in the database. The usage is **GET_DB_POS(point_P, lm_width, inv_lm_width, lm_per_side)**, where:

point_P is a pointer to the location in the database
lm_width is the width of a load module
inv_lm_width is inverse of the width of a load module
lm_per_side is the number of load modules in a row or column of AAM

Defined In:	bx_macros.h	
Called By:	b0_traj_chord bx_trajectory	
Routines Called:	none	
Parameters:	POINT_DATA HWORD REAL_4 HWORD	*point_P lm_width inv_lm_width lm_per_side

B.23 GET_LB_FROM_LM

The GET_LB_FROM_LM macro takes a load module number and calculates the number of the load block that module is in. The usage is **GET_LB_FROM_LM(lm, lb)**, where:

lm is the load module number (0 to 1023)
lb is the load block number (0 to 255)

Defined In:	bx_macros.h
Called By:	b0_new_frame b0_process_round b0_round_fired bx_chord_intersect

Routines Called:	none	
Parameters:	INT_4 INT_4	lm lb

B.24 GLOB

The GLOB macro provides a means by which global variables can be accessed on the Butterfly platform. (The Butterfly takes all of memory_map.h and puts it into a simple C structure.) For the Masscomp, GLOB has no effect — GLOB(x) is defined as x.

Defined In:	ememory_map.h memory_map.h
Called By:	all functions that access global memory
Routines Called:	none
Parameters:	none

B.25 INCR_COMPONENT

The INCR_COMPONENT macro updates a component's word count, polygon count, and vertex count. The usage is **INCR_COMPONENT(incr)**, where *incr* is the count increment.

Defined In:	rcfuncs.c	
Called By:	rcl_component rcl_data	
Routines Called:	none	
Parameters:	WORD	incr

B.26 INIT_MTX

The INIT_MTX macro initializes a 4x3 matrix to the identity matrix. The last column is assumed and zeroes are assumed loaded. This routine is used to initialize the matrices for all static and dynamic vehicles on start-up.

The usage is **INIT_MTX(matrix)**, where *matrix* is the model's transformation matrix.

Defined In:	functions.h	
Called By:	active_area_init	
Routines Called:	none	
Parameters:	REAL_4	matrix

B.27 MALLOC

The MALLOC macro allocates memory. MALLOC calls slave133_malloc if Ballistics is running on a slave board; otherwise it calls the malloc library function.

The usage is **MALLOC(size)**, where *size* is the amount of memory to be allocated.

Defined In:	bx_defines.h	
Called By:	b0_add_traj_table b0_database_info	
Routines Called:	malloc slave133_malloc	
Parameters:	int	size

B.28 NEW_ROUND

The NEW_ROUND macro gets a round from the free list and sets a pointer to it. The usage is **NEW_ROUND(new_round_P)**, where *new_round_P* is the pointer to the round.

Defined In:	bx_macros.h
Called By:	b0_process_round b0_round_fired
Routines Called:	none

Parameters:	ROUND_DATA	*new_round_P

B.29 NEW_STAT_VEH

The NEW_STAT_VEH macro gets a static vehicle from the free list and adds it to a specified load module's list.

The usage is **NEW_STAT_VEH(veh_table_P, new_sv_P)**, where:

veh_table_P is a pointer to the vehicle table
new_sv_P is the pointer to the new vehicle

new_sv_P is set to NULL if no pointers are available (i.e., the maximum number of static vehicles has been reached).

Defined In:	bx_macros.h	
Called By:	b0_add_static_vehicle	
Routines Called:	none	
Parameters:	STRUCT_P_SV STAT_VEH	*veh_table_P *new_sv_P

B.30 OPEN_EXCHANGE

The OPEN_EXCHANGE macro obtains the file descriptors for the input and output channels for CIG-SIM communications. The usage is **OPEN_EXCHANGE.**

Defined In:	functions.h
Called By:	upstart
Routines Called:	dr_is_okay printf
Parameters:	none

B.31 OPEN_FLEA_DATA

The OPEN_FLEA_DATA macro is used by Flea to obtain the file descriptors for the input and output channels for Flea-CIG communications.

The usage is **OPEN_FLEA_DATA(flea_imsg, flea_omsg)**, where:

flea_imsg is a pointer to the input message packet
flea_omsg is a pointer to the output message packet

Defined In:	functions.h	
Called By:	flea	
Routines Called:	sc_pend	
Parameters:	INT_4 INT_4	*flea_imsg *flea_omsg

B.32 PAGE_FORMAT

The PAGE_FORMAT macro handles displays that exceed one page (16 lines). The usage is **PAGE_FORMAT(lines)**, where *lines* is the number of lines in the display.

Defined In:	gos_bal_query.c	
Called By:	gos_bal_query	
Routines Called:	printf scanf	
Parameters:	INT	lines

B.33 poly.* (Poly Processor Macros)

Macros are used by the DTP Command Generator functions to interface to the Runtime Command Library (RCL). These macros call RCL routines that generate the actual commands that are downloaded to the Polygon Graphics Processor.

Each Poly Processor command has one or more supporting macros. The following table lists each Poly Processor macro and identifies its parameters, calling routines, and called routines. It also identifies the Poly Processor command generated by RCL for each macro. Detailed descriptions of the hardware commands are beyond the scope of this document.

Defined In:	rcinclude.h

Called By: see table below

Routines Called: see table below

Parameters: see table below

Macro(parameters)	Poly Processor Command Generated	Called By	Routines Called
poly_ab(alpha_0, beta_0, alpha_1, beta_1)	Alpha Betas	none	rcl_data
poly_bvc(ballistics_bit, local_terrain_bit)	Bounding Volume Component	none	rcl_componen
poly_efs(label, number_of_frames)	Effect Stage	none	rcl_lblcmd
poly_efsr(label, number_of_frames)	Effect Stage Relative	none	rcl_lblcmd
poly_flu()	Flush	dtp_trav1	rcl_command
poly_fsw()	Form Stamp Words	dtp_trav2	rcl_command
poly_gc(ballistics_bit, local_terrain_bit)	Grid Component	none	rcl_component
poly_inf(information_word)	Info Word	none	rcl_data
poly_lmf(matrix_pointer)	Load Matrix Full	none	rcl_command rcl_stuff_data
poly_lsc(x, y, z, w)	Load Screen Constants	none	rcl_command
poly_mmf(matrix_pointer)	Matrix Multiply Full	none	rcl_command, rcl_stuff_data
poly_pc(ballistics_bit, local_terrain_bit)	Poly Component	none	rcl_component
poly_poly(poly_info_word, vertex_list, alpha, beta)	Polygon Entry	none	rcl_data
poly_rm1()	Recall Matrix 1	dtp_trav2	rcl_command
poly_rm2()	Recall Matrix 2	none	rcl_command
poly_rm3()	Recall Matrix 3	none	rcl_command
poly_rm4()	Recall Matrix 4	none	rcl_command
poly_sc(ballistics_bit, local_terrain_bit)	Stamp Component	none	rcl_component
poly_sci(ballistics_bit, local_terrain_bit, stamp_info_word, stamp_half_width, stamp_height)	Stamp Component Incomplete	none	rcl_component, rcl_data
poly_sec(ballistics_bit, local_terrain_bit)	Special Effect Component	none	rcl_component
poly_sm1()	Save Matrix 1	dtp_trav2	rcl_command
poly_sm2()	Save Matrix 2	none	rcl_command
poly_sm3()	Save Matrix 3	none	rcl_command
poly_sm4()	Save Matrix 4	none	rcl_command
poly_stamp(stamp_info_word, stamp_half_width,, stamp_height, stamp_center_x, stamp_center_y, stamp_center_z)	Stamp List Entry	none	rcl_data
poly_tog()	Channel Toggle	dtp_trav2	rcl_command
poly_vtxe(x_value, y_value, z_value)	Vertex List Entry	none	rcl_data
poly_vtxl(index_0, index_1, index_2, index_3)	Vertex List	none	rcl_data

B.34 PRINTD4

The PRINTD4 macro prints a 32-bit word in hexadecimal and decimal format. The address at which to start printing is in the pointer variable *pntr2*. The usage is **PRINTD4()**.

Defined In:	gos_memory.c
Called By:	gos_memory
Routines Called:	printf
Parameters:	none

B.35 PRINTD8

The PRINTD8 macro prints a double in hexadecimal and decimal format. The address at which to start printing is in the pointer variable *pntr2*. The usage is **PRINTD8()**.

Defined In:	gos_memory.c
Called By:	gos_memory
Routines Called:	printf
Parameters:	none

B.36 PRINTHEX4

The PRINTHEX4 macro prints a 32-bit word in hexadecimal format. The address at which to start printing is in the pointer variable *pntr2*. The usage is **PRINTHEX4()**.

Defined In:	gos_memory.c
Called By:	gos_memory
Routines Called:	printf
Parameters:	none

B.37 PRINTHEX8

The PRINTHEX8 macro prints a 64-bit word in hexadecimal format. The address at which to start printing is in the pointer variable *pntr2*. The usage is **PRINTHEX8()**.

Defined In:	gos_memory.c
Called By:	gos_memory
Routines Called:	printf
Parameters:	none

B.38 READ_CLOCK

The READ_CLOCK macro, defined in force_defines.h, is not currently used.

B.39 RESTART_CLOCK

The RESTART_CLOCK macro, defined in force_defines.h, is not currently used.

B.40 ROOM4LABEL

The ROOM4LABEL macro verifies that there is room in the stack to add a label. The usage is **ROOM4LABEL(store, loc)**, where:

store is the location to store the address
loc is the label to set with the AAM location

If the stack does not have room for the label, an error is output.

Defined In:	rcfuncs.c	
Called By:	rcl_lblcmd rcl_set_cntlbl rcl_set_label	
Routines Called:	ERRMSG printf	
Parameters:	WORD	*store

WORD	m

B.41 ROOMCHECK

The ROOMCHECK macro verifies that there is enough space for a command. The usage is **ROOMCHECK(name, wd_cnt)**, where:

name is a pointer to the routine name
wd_cnt is the number of command WORDs

The function outputs an error if space is insufficient.

Defined In: rcfuncs.c

Called By: rcl_command
rcl_component
rcl_lblcmd

Routines Called: ERRMSG

Parameters:		
	char	*name
	WORD	wd_cnt

B.42 SET_OUT_BITS

The SET_OUT_BITS macro, defined in definitions.h, is not currently used.

B.43 SET_OUT_M2BITS

The SET_OUT_M2BITS macro, defined in definitions.h, is not currently used.

B.44 SYSERR

The SYSERR macro adds an error message to the output buffer and ends processing of input messages by pointing to a dummy end statement. The usage is **SYSERR(error, state)**, where:

error is the error message
state is the current state of the CIG

Defined In: functions.h

Called By: cig_config
db_mcc_setup

	file_control get_msg_2d hw_test open_dbase simulation upstart	
Routines Called:	none	
Parameters:	INT_2 INT_2	error state

B.45 TORAD

The TORAD macro converts an angle into radians. The usage is **TORAD(angle)**, where *angle* is the angle in degrees. The routine multiplies the given angle by 0.017453292.

Defined In:	concat_mtx.c flea_decode_data.c flea_encode_data.c flea_init_cig_sw.c flea_update_pos.c gos_flea_options.c gos_model.c simulation.c update_fov.c upstart.c	
Called By:	concat_mtx flea_decode_data flea_encode_data flea_init_cig_sw flea_update_pos gos_flea_options gos_model simulation update_fov upstart	
Routines Called:	none	
Parameters:	INT	angle

B.46 toradians

The toradians macro converts an angle into radians. The usage is **toradians(angle)**, where angle is the angle in degrees. The routine multiplies the given angle by 0.017453293.

Defined In:	make_bbn.c	
Called By:	rotate_x rotate_y rotate_z	
Routines Called:	none	
Parameters:	INT	angle

B.47 TRIGGER_FORCE

The TRIGGER_FORCE macro puts a command into the Force front-end control register (FE_CONTROL). The value in this register tells the forcetask what command is to be performed. The usage is **TRIGGER_FORCE**.

Defined In:	functions.h
Called By:	gsp_load
Routines Called:	none
Parameters:	none

B.48 WAIT_FORCE

The WAIT_FORCE macro polls the ready bit (FRONT_RDY_MASK) in the Force front-end control (FE_CONTROL) register, waiting for it to be 0. The usage is **WAIT_FORCE**.

Defined In:	functions.h
Called By:	DOWNLOAD_DATA gsp_load

Routines Called: printf

Parameters: none

B.49 XCLOSE

The XCLOSE macro is defined as the system call "close" for the 120T CIG MVME133, and "fclose" for the Butterfly.

Defined In: definitions.h

Called By:
get_lm
gsp_load
load_dbase
open_dbase
open_ded
rowcol_rd
simulation
sload

Routines Called:
close
fclose

Parameters: none

L.50 XLSEEK

The XLSEEK macro is defined as the system call "lseek" for the 120T CIG MVME133, and "flseek" for the Butterfly.

Defined In: definitions.h

Called By:
download_bvols
get_lm
getlmdp
getside
load_dbase
open_dbase
open_ded

Routines Called:
flseek
lseek

Parameters: none

B.51 XOPEN

The XOPEN macro is defined as the system call "open" for the 120T CIG MVME133, and "fopen" for the Butterfly.

Defined In: definitions.h

Called By:
flea_init_cig_sw
get_lm
gsp_load
open_dbase
open_ded
read_configfile
rowcol_rd
sload

Routines Called:
fopen
open

Parameters: none

B.52 XREAD

The XREAD macro is defined as the system call "read" for the 120T CIG MVME133, and "fread" for the Butterfly.

Defined In: definitions.h

Called By:
download_bvols
get_char
get_lm
getlmdp
getside
gsp_load
load_dbase
open_dbase
open_ded

Routines Called:
fread
read

Parameters: none

B.53 XWRITE

The XWRITE macro is defined as the system call "write" for the 120T CIG MVME133, and "fwrite" for the Butterfly.

Defined In: definitions.h

Called By: none

Routines Called: fwrite
write

Parameters: none

APPENDIX C. OPERATING SYSTEM SERVICE CALLS

This appendix provides brief descriptions of the various operating system calls and standard C library routines used by the CIG Real-Time software.

C.1 Special OS Service Libraries

The following table describes the system-level service routines used by the CIG Real-Time software.

Routine	Description	Called By
read_watch()	Gets the cumulative number of 500 uS ticks. Returns the number as *watch_count.*	local_terrain, simulation
start_watch()	Determines which CPU board it is executing on, sets up the timer registers appropriately, clears the stopwatch storage areas, starts the timer, and enables the interrupts. Returns *board.*	simulation
stop_watch()	Gets the cumulative number of 500 (100)uS ticks, stops the timer, and turns the timer interrupts off. Returns *watch_count.*	simulation
sysrup_off()	Ignores the system/frame interrupt by moving the address of a null interrupt service routine into the 68010 exception vector space.	simulation, dtp_emu, gos_model, gos_system, dcode_dr11w, *gos_single_step, s_step*
sysrup_on(mailbox_ptr, message)	Enables system/frame interrupts by moving the address of the interrupt service routine in to the 68010 exception vector space. Wakes up a pending routine by moving the calling task's mailbox address and the message to be returned to locations known to the isr.	simulation, dtp_emu, gos_model, gos_system, s_step

C.2 Task Management (sc_*) Routines

The following table describes the routines that handle intertask mailbox communication and the creation and deletion of tasks and queues. These routines are standard Ready Systems' VRTX C interface libraries.

Routine	Description	Called By
sc_accept(mailbox_ptr, error_ptr)	Clears messages from the specified mailbox.	simulation
sc_lock()	Locks a queue to prevent concurrent use.	dr_is_okay, mx_open, mx_peek, mx_push, mx_skip
sc_pend(mailbox_ptr, timeout, error_ptr)	Waits for a message to be posted to the specified mailbox.	cig_config, simulation, rowcol_rd, local_terrain, gos_flea_if, gossip, flea, flea_init_cig_sw, OPEN_FLEA_DATA, EXCHANGE_DATA, EXCHANGE_DATA_SIM, EXCHANGE_FLEA_DATA
sc_post(mailbox_ptr, message, error_ptr)	Posts a message to the specified mailbox.	cig_config, db_mcc_setup, simulation, upstart, rowcol_rd, local_terrain, gos_atp, gos_flea_if, gos_fly, flea_init_cig_sw, DART_ENQUEUE, EXCHANGE_DATA, EXCHANGE_DATA_SIM, EXCHANGE_FLEA_DATA
sc_qcreate(queue_id, size, error_ptr)	Creates a system queue of the specified size.	qassign
sc_qinquiry(queue_id, count_ptr, error_ptr)	Counts the entries in the specified queue.	dr_is_okay
sc_qpend(queue_id, timeout, error_ptr)	Removes messages from the specified queue.	dr_is_okay
sc_tcreate(task_entry, task_id, task_priority, error_ptr)	Creates a system task.	tassign
sc_tdelete(task_id, priority_code, error_ptr)	Deletes a task from the system.	apinit
sc_unlock()	Unlocks a locked queue.	dr_is_okay, mx_open, mx_peek, mx_push, mx_skip

C.3 Standard C Runtime Libraries

The following table identifies the standard C system calls, input/output routines, and runtime libraries used by the CIG Real-Time software.

Routine	Description	Called By
atof	Converts a string to double.	REAL4_fscanf
atoi	Converts a string to int.	main (in upstart), main (in rowcol_rd), main (in local_terrain)
bzero	Places a specified length of 0 bytes into a specified string.	main (in upstart)
calloc	Allocates memory and initializes to zero.	init_configtree, cig_2d_setup
close	Closes a file.	XCLOSE, read_configfile, file_control, gos_memory, flea_init_cig_sw
cmd	Sends a command to sio.	getch
cos	Calculates a cosine.	update_fov, rotate_x, rotate_y, rotate_z, gos_flea_options, gos_model, flea_encode_data, flea_update_pos, flea_init_cig_sw
create_sz	Creates a file with a specified size.	file_control, gos_memory
fclose	Closes an I/O stream.	XCLOSE
fflush	Writes all currently buffered characters in an output stream.	get_char (Butterfly version), unbf_getchar (Butterfly version)
flseek	Moves the read/write pointer.	XLSEEK
fopen	Opens an I/O stream.	XOPEN
fread	Reads a specified number of bytes.	XREAD
free	Frees allocated memory.	free_configtree, download_bvols, ,oad_dbase, open_dbase, simulation, cig_2d_setup, b0_add_traj_table, bx_reset
fwrite	Writes to a file.	XWRITE
lseek	Moves the read/write pointer.	XLSEEK, file_control, flea_init_cig_sw
open	Opens a file.	XOPEN, file_control, gos_memory, flea_init_cig_sw
outhexl	Outputs a hex value to stdout.	b0_delete_static_vehicle, b0_traj_entry
printf	Writes to stdout.	(used extensively throughout system)
puts	Writes to stdout.	b0_delete_static_vehicle, b0_traj_entry
read	Reads a file.	XREAD, file_control, gos_memory, flea_init_cig_sw
rsec	Reads multiple sectors from disk.	file_control

scanf	Reads from stdin.	dtp_emu, gos_120tx, gos_bal_query, gos_db_query, gos_flea_if, gos_flea_options, gos_fly, gos_locate, gos_memory, gos_model, gos_system, gossip, PAGE_FORMAT
sin	Calculates a sine.	update_fov, rotate_x, rotate_y, rotate_z, upstart, gos_flea_options, gos_model, flea_encode_data, flea_update_pos, flea_init_cig_sw
sqrt	Calculates a square root.	dtp_emu, gos_model
strcmp	Compares two strings.	find_fn, setup_comp_start, process_command
strcpy	Copies a string.	apinit, confignode_setup, file_control, bootup_slave133
strlen	Length of string.	file_control, open_dbase, open_ded, setup_define_string, setup_text, gossip, flea_init_cig_sw
system	Executes a shell command.	find_fn, file_control, bootup_slave133, dr, gsp_load
tan	Calculates a tangent.	update_fov
unbf_getchar	Performs an unbuffered getchar.	dtp_emu, cal, gos_120tx, gos_atp, gos_bal_query, gos_db_query, gos_flea_if, gos_flea_options, gos_fly, gos_memory, gos_model, gos_system, gossip, s_step
write	Writes a specified number of bytes.	XWRITE, gos_memory, file_control, cig_config, EXCHANGE_DATA, EXCHANGE_DATA_SIM

APPENDIX D. GLOSSARY OF TERMS AND ABBREVIATIONS

2-D	Two-dimensional.
AAM	Active area memory. Memory that contains the currently viewable database and models. AAM contains 256 terrain load modules (16 rows by 16 columns). This provides a 3500-meter viewing range, plus a 500-meter buffer, in each direction. If load module blocking is enabled, AAM is effectively quadrupled.
AGL	Above ground level. If AGL processing is enabled (via the MSG_AGL_SETUP message), the simulated vehicle's altitude above ground level is calculated and returned to the Simulation Host every frame.
ASID	Application-specific identification data. ASIDs are used to add unique data (e.g., bumper numbers, smoke plume, dust cloud, etc.) to a model.
aspect ratio	The ratio of the sides (width:height) of the viewport. This is assumed to be 1.
BVME	A VME board that interfaces with the Butterfly computer.
bvol	Bounding volume. The volume of the bounding box that is used to completely enclose an object in the simulation environment.
centroid	The theoretical "center" of an object, around which the object is rotated. The centroid's coordinates are the averages of the corresponding coordinates of a given set and, for a given planar or three-dimensional figure (such as a triangle or sphere), correspond to the center of mass of a thin plate of uniform thickness and consistency or a body of uniform consistency having the same boundary.
channel	A connection to a viewport. One channel may have multiple graphics paths.
CIG	Computer Image Generation System. The process of generating a 3-D, perspectively accurate scene via a computer.
clipping	Removing back-facing polygons or parts of polygons that lie partially outside the viewing pyramid.
conditional node	A node in the configuration tree that causes a branch into one of two traversal paths based on some runtime condition.
configuration tree	A structure that defines the relationship between each physical component of the simulation vehicle and the location of the viewports.
data message	Smallest data component of a packet buffer.

data message header	A message that describes the contents of a data message.
DED	Dynamic Elements Database.
double-buffer memory	Memory that contains the dynamic models built by the real-time software and processed by the hardware. Dual buffering allows for one buffer to be used by the hardware while the other is being updated by the software. The buffer used for each purpose switches each frame, so the hardware is always using the buffer updated by the software during the previous frame.
downloading	The process of transferring data from the Simulation Host to the CIG.
DR11-W	A Digital Equipment Corp. standard interface that enables the Simulation Host and the CIG processor to communicate at a high transmission rate.
DTP	Data Traversal Processor.
dynamic vehicle	A vehicle whose position and orientation is redefined in every frame sent by the Simulation Host.
false child	The configuration tree node branched to from a conditional node if the runtime conditions is false.
fov	Field of view. The volume of space which encompasses all objects that are visible from a specific viewpoint and view angle.
frame	Information displayed on a television monitor for 33.3 milliseconds (at 30 Hz) or 66.6 milliseconds (at 15 Hz).
frame event	An interrupt signal given by the hardware.
frame rate	The rate at which a new image is created and displayed on the screen.
frame time	The amount of time each frame is displayed.
graphics path	A window on a viewport. The 120T has one graphics path per viewport. The 120TX may have two or four, depending on the resolution. Graphics path parameters are the viewport parameters that are used to load the hardware.
GSP	Graphics System Processor. The TMS34010 graphics processor on the MPV board that generates and controls 2-D graphics.
graphics processor	First board in the graphics pipeline that processes 3-D data and converts it into 2-D screen space for the tiler, based on the input of graphics processor commands. Also called the poly processor.
heading	The direction the viewer is pointing.

hull transformation	Description of the position and orientation of the base of a vehicle.
Hz	Hertz; cycles per second.
load module	A unit of terrain in the terrain database, measuring 500 meters by 500 meters. Data is brought into active area memory in whole load modules only.
load module block	A structure containing four load modules (two rows by two columns, for a total size of 1000 meters by 1000 meters). Blocking load modules doubles the viewing range and quadruples the amount of terrain that can be loaded into active area memory.
lod	Level of detail. The selective reduction of model detail (polygon count) or texture map detail based on distance from the viewer.
lookup table	A table used to convert color-map addresses into the actual color values displayed.
matrix	A rectangular array of elements arranged in rows and columns.
matrix node	A node in the configuration tree that contains a transformation matrix. The matrices in each node in a traversal path are concatenated to generate the view of the world for the viewport represented by that path.
MCC	Management, Command, and Control. The computer on the simulation network that monitors and controls the entire simulation exercise.
model	Generally used to refer to models of arbitrary, three-dimensional objects such as buildings and vehicles.
model space	The coordinate system used to define and build a particular model. The vehicle's centroid is defined as location (0,0,0).
MPV	Micro Processor Video. The last board in the graphics pipeline in a 120TX system.
My_Vehicle	The simulation vehicle.
object	All simulated models: vehicles, hidden obstacles, etc.
overlay	A two-dimensional view that is displayed on a viewport on top of the three-dimensional view of the terrain.
packet buffer	Several data messages grouped together that describe one frame time.
pitch	The angle at which the viewer is looking up or down.
pixel	Picture element. The smallest addressable element on a video screen.

Poly Processor	See graphics processor.
polygon	A closed, planar figure bounded by straight lines and consisting of three or four vertices.
real-time	The ability to respond rapidly, frequently, or both to an event or transaction. Also refers to the software that is used to run real-time operations.
roll	The angle which measures the amount of rotation along the viewing vector (tilt).
rotation	The process by which coordinates are rotated around a particular axis. Used to define the direction of the viewing window.
rotation matrix	A means of specifying orientation.
RCL	Runtime command library. A set of routines used to generate hardware commands for the DTP and the Poly Processor.
RTS	Rotation translation scale.
scaling	The process by which an object's coordinates are changed to effectively enlarge, reduce, or skew the object in a particular direction.
SIM	The Simulation Host computer. The computer that controls the simulated vehicle's behavior.
simulation	The process that involves a computerized model of specific, significant features of some physical or logical system or environment.
simulation vehicle	The vehicle represented by a simulated viewpoint. Also called simulated vehicle or My_Vehicle.
simulator	A simulation unit consisting of a Simulation Host, a CIG, one or more monitors, and the vehicle controls. Also called a Vehicle Simulator Unit.
static vehicle	A vehicle with no anticipated movement, tracked only when its status changes.
T&C	Timing and Control. Board that controls all CIG synchronization and timing.
terrain database	The database on the CIG that contains the polygons that describe the simulation terrain and all objects (houses, trees, etc.) in it.
translation	The process by which coordinates are "moved" from one location to another.

transformation	A combination of translations and rotations that convert the coordinates of a point in one coordinate system into coordinates in another coordinate system.
transformation matrix	A matrix used to describe the position and orientation of an object.
true child	The configuration tree node branched to from a conditional node if the runtime conditions is true.
vector	A straight line with a specific direction.
vertex	A point in space, the termination point of a line, or the intersection point of two or more lines.
viewpoint	The direction of view from the user's eye to the target or object being viewed.
viewport	A display screen connected to the CIG. Each viewport simulates the view of the world from a specific window of the simulated vehicle.
viewport parameters	The screen resolution, viewing range, near plane, field-of-view angles, level-of-detail multiplier, and aspect ratio (currently not used) of a viewport.
viewspace	The area that falls within the field of view of a viewport.
VME	Versa Module European. An industry-standard bus.
world space	The absolute coordinate system used to define the simulation area. A three-dimensional space fixed relative to the world. Location (0,0) is the southwest corner of the database.

APPENDIX E. CROSS-REFERENCE TABLES

This appendix contains the following cross-reference tables:

E.1 CSUs (source files) mapped to CSCs.
E.2 Data type names mapped to location of typedef.
E.3 Function names mapped to source file location, with section numbers.
E.4 Macro names mapped to source file location, with section numbers.

E.1 CSUs Mapped To CSCs

The following list shows every CSU (.c or .asm file) in the CIG Real-Time CSCI, and identifies the CSC to which it belongs. The CSUs are listed in alphabetical order.

CSU	CSC
aa_init.c	UPSTART (Real-Time Processing component)
aam_manager.c	UPSTART (Viewport Configuration component)
b0_aam_centroid.c	BALLISTICS (Interface Messaging component)
b0_aam_sw_corner.c	BALLISTICS (Interface Messaging component)
b0_add_static_vehicle.c	BALLISTICS (Interface Messaging component)
b0_add_traj_table.c	BALLISTICS (Interface Messaging component)
b0_bal_config.c	BALLISTICS (Interface Messaging component)
b0_bvol_entry.c	BALLISTICS (Interface Messaging component)
b0_cancel_round.c	BALLISTICS (Interface Messaging component)
b0_cig_frame_rate.c	BALLISTICS (Interface Messaging component)
b0_database_info.c	BALLISTICS (Interface Messaging component)
b0_delete_static_vehicle.c	BALLISTICS (Interface Messaging component)
b0_delete_traj_table.c	BALLISTICS (Interface Messaging component)
b0_dummy.c	BALLISTICS (Interface Messaging component)
b0_error_detected.c	BALLISTICS (Interface Messaging component)
b0_inapp_message.c	BALLISTICS (Interface Messaging component)
b0_lm_read.c	BALLISTICS (Interface Messaging component)
b0_model_directory.c	BALLISTICS (Interface Messaging component)
b0_model_entry.c	BALLISTICS (Interface Messaging component)
b0_new_frame.c	BALLISTICS (Interface Messaging component)
b0_print.c	BALLISTICS (Interface Messaging component)
b0_process_chord.c	BALLISTICS (Interface Messaging component)
b0_process_round.c	BALLISTICS (Interface Messaging component)
b0_round_fired.c	BALLISTICS (Interface Messaging component)
b0_state_control.c	BALLISTICS (Interface Messaging component)
b0_status_request.c	BALLISTICS (Interface Messaging component)
b0_traj_chord.c	BALLISTICS (Interface Messaging component)
b0_traj_entry.c	BALLISTICS (Interface Messaging component)
b0_undefined_message.c	BALLISTICS (Interface Messaging component)
bal_get_db_pos.c	LOCAL_TERRAIN
bal_get_lm_grid.c	ROWCOL_RD
bal_routines.c	UPSTART (Real-Time Processing component)
bbnctype.c	UPSTART (Viewport Configuration component)
bit_blt.c	UPSTART (2-D Overlay Compiler component)
bus_error.asm	UPSTART (Real-Time Processing component)
bx147_main.c	BALLISTICS (Mainline component)
bx_bvol_int.c	BALLISTICS (Intersection Calculations component)
bx_chord_intersect.c	BALLISTICS (Intersection Calculations component)
bx_functions.c	BALLISTICS (Intersection Calculations component)
bx_get_lm_data.c	BALLISTICS (Intersection Calculations component)
bx_get_lm_grid.c	BALLISTICS (Intersection Calculations component)
bx_init.c	BALLISTICS (Mainline component)
bx_model_int.c	BALLISTICS (Intersection Calculations component)
bx_poly_int.c	BALLISTICS (Intersection Calculations component)
bx_reset.c	BALLISTICS (Intersection Calculations component)
bx_task.c	BALLISTICS (Mainline component)
bx_trajectory.c	BALLISTICS (Intersection Calculations component)
cal.c	UPSTART (Real-Time Processing component)
cig_2d_setup.c	UPSTART (2-D Overlay Compiler component)

CSU	CSC
cig_comp_2d.c	UPSTART (2-D Overlay Compiler component)
cig_config.c	UPSTART (Viewport Configuration component)
cig_getm_2d.c	UPSTART (2-D Overlay Compiler component)
cig_link_2d.c	UPSTART (2-D Overlay Compiler component)
comp.c	UPSTART (2-D Overlay Compiler component)
concat_mtx.	UPSTART (Viewport Configuration component)
confignode_setup.c	UPSTART (Viewport Configuration component)
data_type.c	FORCE
db_mcc_setup.c	UPSTART (Real-Time Processing component)
debug_initdr.c	UPSTART (Real-Time Processing component)
ded_model_trace.c	UPSTART (Real-Time Processing component)
download_bvols.c	UPSTART (Real-Time Processing component)
dr.c	UPSTART (Real-Time Processing component)
draw_line.c	UPSTART (2-D Overlay Compiler component)
dtp_compiler	UPSTART (DTP Command Generator component)
dtp_emu.c	GOSSIP
dtp_funcs.s	UPSTART (DTP Command Generator component)
dtp_trav1.c	UPSTART (DTP Command Generator component)
dtp_trav2.c	UPSTART (DTP Command Generator component)
exception.asm	FORCE
file_control.c	UPSTART (Real-Time Processing component)
fill_tree.c	UPSTART (Viewport Configuration component)
find_fn.c	UPSTART (Real-Time Processing component)
flea.c	FLEA
flea_decode_data.c	FLEA
flea_encode_data.c	FLEA
flea_init_cig_sw.c	FLEA
flea_update_pos.c	FLEA
force.asm	FORCE
forcetask.c	FORCE
fxbvtofl.c	UPSTART (Real-Time Processing component)
generic_lm.c	ROWCOL_RD
get_thing.c	UPSTART (2-D Overlay Compiler component)
getch.c	UPSTART (Viewport Configuration component)
gos_120tx.c	GOSSIP
gos_atp.c	GOSSIP
gos_bal_query.c	GOSSIP
gos_db_query.c	GOSSIP
gos_dr11_query.c	GOSSIP
gos_flea_if.c	GOSSIP
gos_flea_options.c	GOSSIP
gos_fly.c	GOSSIP
gos_locate.c	GOSSIP
gos_memory.c	GOSSIP
gos_model.c	GOSSIP
gos_polys.c	GOSSIP
gos_system.c	GOSSIP
gossip.c	GOSSIP
gsp_io.c	FORCE
gsp_load.c	UPSTART (Real-Time Processing component)
gun_overlays.c	UPSTART (Real-Time Processing component)
hw_test.c	UPSTART (Real-Time Processing component)
init_stuff.c	UPSTART (2-D Overlay Compiler component)
load_dbase.c	UPSTART (Real-Time Processing component)
load_modules.c	ROWCOL_RD
loc_ter.c	ROWCOL_RD
make_bbn.c	UPSTART (Real-Time Processing component)
mat_dump.c	UPSTART (Viewport Configuration component)

CSU	CSC
mkcal.c	UPSTART (Real-Time Processing component)
mkmtx_nt.c	UPSTART (Real-Time Processing component)
model_mtx.c	UPSTART (Real-Time Processing component)
mx_error.c	BALLISTICS (Message Queue Processing component)
mx_open.c	BALLISTICS (Message Queue Processing component)
mx_peek.c	BALLISTICS (Message Queue Processing component)
mx_push.c	BALLISTICS (Message Queue Processing component)
mx_skip.c	BALLISTICS (Message Queue Processing component)
mx_wcopy.c	BALLISTICS (Message Queue Processing component)
nmi_type.c	FORCE
open_dbase.c	UPSTART (Real-Time Processing component)
open_ded.c	UPSTART (Real-Time Processing component)
oval_rect.c	UPSTART (2-D Overlay Compiler component)
overlay_setup.c	UPSTART (Viewport Configuration component)
poll_ready.c	FORCE
poly.c	UPSTART (2-D Overlay Compiler component)
proc_cmd.c	UPSTART (2-D Overlay Compiler component)
process_vflags.c	UPSTART (Viewport Configuration component)
process_vppos.c	UPSTART (Viewport Configuration component)
rcfuncs.c	UPSTART (DTP Command Generator component)
read_configfile.c	UPSTART (Viewport Configuration component)
read_stuff.c	FORCE
rowcol_rd.c	ROWCOL_RD
rtt.c	RTT
simulation.c	UPSTART (Real-Time Processing component)
slave133_functions.c	BALLISTICS (Mainline component)
stdio.c	UPSTART (Real-Time Processing component)
string.c	UPSTART (2-D Overlay Compiler component)
support.c	UPSTART (Real-Time Processing component)
test_gsp.c	FORCE
text.c	UPSTART (2-D Overlay Compiler component)
update_fov.c	UPSTART (Viewport Configuration component)
update_rez.c	UPSTART (Viewport Configuration component)
upstart.c	UPSTART (Real-Time Processing component)
vec_dump.c	UPSTART (Viewport Configuration component)
viewport_setup.c	UPSTART (Viewport Configuration component)
vt100.c	GOSSIP
window.c	UPSTART (2-D Overlay Compiler component)

E.2 Data Type Names Mapped To Typedefs

The following list shows the special data types used throughout the real-time software, and identifies the file that provides the type definition. The type names are listed in alphabetical order.

Data Type	Typedef Location
ALLOC_POLY	/120/tx/include/struct_2d.h
ASID_OMODEL	/120tx/include/structures.h
ASID_SHOW_EFF	/120tx/include/structures.h
B1BBOX2D	/120tx/include/dgi_stdg.h
B1BBOX3D	/120tx/include/dgi_stdg.h
B1HSL	/120tx/include/dgi_stdg.h
B1HSLO	/120tx/include/dgi_stdg.h
B1MTX4X3	/120tx/include/dgi_stdg.h
B1MTX4X4	/120tx/include/dgi_stdg.h
B1P2D	/120tx/include/dgi_stdg.h
B1P3D	/120tx/include/dgi_stdg.h
B1P4D	/120tx/include/dgi_stdg.h
B1RGB	/120tx/include/dgi_stdg.h
B1RGBO	/120tx/include/dgi_stdg.h
BOOLEAN	/120tx/include/dgi_stdc.h
BVOL_ENTRY	/120tx/include/rtdb_struct.h
BYTE	/120tx/include/dgi_stdc.h
CAL_OVRLY	/120tx/include/structures.h
CATALOG_TABLE_STRUCT	/120tx/include/rtdb_struct.h
CHAN_CNST	/120tx/include/structures.h
CHAN_SETCMD	/120tx/include/structures.h
CHORD_DATA	/120tx/include/structures.h
CLR_FLAGS	/120tx/source/source/load_dbase.c
CMD	/120tx/include/structures.h
CMDR_OVRLY	/120tx/include/structures.h
COMMAND_LINE1	/120tx/include/structures.h
COMMAND_LINE2	/120tx/include/structures.h
CONFIGURATION_NODE	/120tx/include/configtree_str.h
DB_DIR_ENTRY	/120tx/include/rtdb_struct.h
DB_HDR_DBASE_DATA	/120tx/include/rtdb_struct.h
DB_HDR_LMARKS_DATA	/120tx/include/rtdb_struct.h
DB_HDR_OFLOW_DATA	/120tx/include/rtdb_struct.h
DB_TAG_STRUCT	/120tx/include/rtdb_struct.h
DB_VERSION_STRUCT	/120tx/include/rtdb_struct.h
DGI_TO_LABS_MSGS	/120/tx/include/ci_bfly.h
DTP_CMND_INF	/120tx/source/source/ded_model_trace.c
EDGE_FLG	/120tx/include/definitions.h
EO_EFFECTS	/120tx/include/structures.h
EO_OVRLY	/120tx/include/structures.h
FAKE_DV	/120tx/include/structures.h
FIX_BVOL_ENTRY	/120tx/include/rtdb_struct.h
FORCE_INTERFACE	/120tx/force/mbx.h
FOV	/120tx/include/sim_cig_if.h
FOV_VECTORS	/120tx/include/configtree_str.h
FOVTT	/120tx/include/structures.h
GENLM	/120tx/source/source/generic_lm.c
GM_DIR_ENTRY_DATA	/120tx/include/rtdb_struct.h
GM_DIR_ENTRY_NAME	/120tx/include/rtdb_struct.h

Data Type	Typedef Location
GRAPHICS_PATH_PARAMETERS	/120tx/include/configtree_str.h
GRID_COMP_DEF	/120tx/include/structures.h
GRID_LOC	/120tx/include/rtdb_struct.h
GUN_B	/120tx/include/structures.h
GUN_B_LSIDE	/120tx/include/structures.h
GUN_B_RSIDE	/120tx/include/structures.h
GUNNER_OVRLY	/120tx/include/structures.h
HWORD	/120tx/include/dgi_stdc.h
I2BBOX2D	/120tx/include/dgi_stdg.h
I2BBOX3D	/120tx/include/dgi_stdg.h
I2HSL	/120tx/include/dgi_stdg.h
I2HSLO	/120tx/include/dgi_stdg.h
I2MTX4X3	/120tx/include/dgi_stdg.h
I2MTX4X4	/120tx/include/dgi_stdg.h
I2P2D	/120tx/include/dgi_stdg.h
I2P3D	/120tx/include/dgi_stdg.h
I2P4D	/120tx/include/dgi_stdg.h
I2RGB	/120tx/include/dgi_stdg.h
I2RGBO	/120tx/include/dgi_stdg.h
I4BBOX2D	/120tx/include/dgi_stdg.h
I4BBOX3D	/120tx/include/dgi_stdg.h
I4HSL	/120tx/include/dgi_stdg.h
I4HSLO	/120tx/include/dgi_stdg.h
I4MTX4X3	/120tx/include/dgi_stdg.h
I4MTX4X4	/120tx/include/dgi_stdg.h
I4P2D	/120tx/include/dgi_stdg.h
I4P3D	/120tx/include/dgi_stdg.h
I4P4D	/120tx/include/dgi_stdg.h
I4RGB	/120tx/include/dgi_stdg.h
I4RGBO	/120tx/include/dgi_stdg.h
INT_2	/120tx/include/dgi_stdc.h
INT_4	/120tx/include/dgi_stdc.h
LABS_TO_DGI_MSGS	/120/tx/include/ci_bfly.h
LM_CALL1	/120tx/include/structures.h
LM_CALL2	/120tx/include/structures.h
LM_H	/120tx/include/rtdb_struct.h
LM_STATS	/120tx/include/rtdb_struct.h
LMS_DATA	/120tx/include/structures.h
LT_BVOL_ENTRY	/120tx/include/sim_cig_if.h
LT_POLY_ENTRY	/120tx/include/sim_cig_if.h
M1_GUN_OVERLAY	/120tx/include/structures.h
M2_GUN_OVERLAY	/120tx/include/structures.h
MAT_UNIT	/120tx/include/structures.h
MESSAGE_HEADER	/120/tx/include/mx_defines.h
MODEL_TABLE_STRUCT	/120tx/include/rtdb_struct.h
MSG_1ROTATION	/120tx/include/sim_cig_if.h
MSG_2D_SETUP	/120tx/include/sim_cig_if.h
MSG_3ROTATIONS	/120tx/include/sim_cig_if.h
MSG_ADD_TRAJ_TABLE	/120tx/include/sim_cig_if.h
MSG_AGL	/120tx/include/sim_cig_if.h
MSG_AGL_SETUP	/120tx/include/sim_cig_if.h
MSG_AIRVEH_STATE	/120tx/include/sim_cig_if.h
MSG_AMMO_DEFINE	/120tx/include/sim_cig_if.h
MSG_ASID_OTHERVEH_STATE	/120tx/include/sim_cig_if.h
MSG_ASID_SHOW_EFFECT	/120tx/include/sim_cig_if.h
MSG_ASID_STATICVEH_STATE	/120tx/include/sim_cig_if.h
MSG_B0_AAM_CENTROID	/120tx/include/bx_messages.h
MSG_B0_AAM_SW_CORNER	/120tx/include/bx_messages.h

Data Type	Typedef Location
MSG_B0_ADD_STATIC_VEHICLE	/120tx/include/bx_messages.h
MSG_B0_ADD_TRAJ_TABLE	/120tx/include/bx_messages.h
MSG_B0_BAL_CONFIG	/120tx/include/bx_messages.h
MSG_B0_BVOL_ENTRY	/120tx/include/bx_messages.h
MSG_B0_CANCEL_ROUND	/120tx/include/bx_messages.h
MSG_B0_CIG_FRAME_RATE	/120tx/include/bx_messages.h
MSG_B0_DATABASE_INFO	/120tx/include/bx_messages.h
MSG_B0_DELETE_STATIC_VEHICLE	/120tx/include/bx_messages.h
MSG_B0_DELETE_TRAJ_TABLE	/120tx/include/bx_messages.h
MSG_B0_LM_READ	/120tx/include/bx_messages.h
MSG_B0_MODEL_DIRECTORY	/120tx/include/bx_messages.h
MSG_B0_MODEL_ENTRY	/120tx/include/bx_messages.h
MSG_B0_NEW_FRAME	/120tx/include/bx_messages.h
MSG_B0_PRINT	/120tx/include/bx_messages.h
MSG_B0_PROCESS_CHORD	/120tx/include/bx_messages.h
MSG_B0_PROCESS_ROUND	/120tx/include/bx_messages.h
MSG_B0_ROUND_FIRED	/120tx/include/bx_messages.h
MSG_B0_STATE_CONTROL	/120tx/include/bx_messages.h
MSG_B0_TRAJ_CHORD	/120tx/include/bx_messages.h
MSG_B0_TRAJ_ENTRY	/120tx/include/bx_messages.h
MSG_B1_GLOBAL_ADDR	/120tx/include/bx_messages.h
MSG_B1_HIT_RETURN	/120tx/include/bx_messages.h
MSG_B1_MISS	/120tx/include/bx_messages.h
MSG_B1_ROUND_POSITION	/120tx/include/bx_messages.h
MSG_B1_STATUS_RETURN	/120tx/include/bx_messages.h
MSG_CANCEL_ROUND	/120tx/include/sim_cig_if.h
MSG_CIG_CTL	/120tx/include/sim_cig_if.h
MSG_CREATE_CONFIGNODE	/120tx/include/sim_cig_if.h
MSG_DELETE_TRAJ_TABLE	/120tx/include/sim_cig_if.h
MSG_DR11_PKT_SIZE	/120tx/include/sim_cig_if.h
MSG_EO	/120tx/include/sim_cig_if.h
MSG_FILE_DESCR	/120tx/include/sim_cig_if.h, sysdefs.h, sysdefs2.h
MSG_FILE_STATUS	/120tx/include/sim_cig_if.h, sysdefs.h, sysdefs2.h
MSG_FILE_XFER	/120tx/include/sim_cig_if.h, sysdefs.h, sysdefs2.h
MSG_GEN_CONFIGTREE	/120tx/include/sim_cig_if.h
MSG_GENVEH_STATE	/120tx/include/sim_cig_if.h
MSG_GUN_OVERLAY	/120tx/include/sim_cig_if.h
MSG_HDR	/120tx/include/sim_cig_if.h
MSG_HIT	/120tx/include/sim_cig_if.h
MSG_HIT_RETURN	/120tx/include/sim_cig_if.h
MSG_HPRXYZS_MATRIX	/120tx/include/sim_cig_if.h
MSG_LASER_RETURN	/120tx/include/sim_cig_if.h
MSG_LOCAL_TERRAIN	/120tx/include/sim_cig_if.h
MSG_LT_PIECE	/120tx/include/sim_cig_if.h
MSG_M1VEH_STATE	/120tx/include/sim_cig_if.h
MSG_M2VEH_STATE	/120tx/include/sim_cig_if.h
MSG_MISS	/120tx/include/sim_cig_if.h
MSG_OBSCURE	/120tx/include/sim_cig_if.h
MSG_OTHERVEH_STATE	/120tx/include/sim_cig_if.h
MSG_OVERLAY_SETUP	/120tx/include/sim_cig_if.h
MSG_PASS_BACK	/120tx/include/sim_cig_if.h
MSG_PASS_ON	/120tx/include/sim_cig_if.h
MSG_PROCESS_ROUND	/120tx/include/sim_cig_if.h
MSG_REQUEST_LASER_RANGE	/120tx/include/sim_cig_if.h
MSG_ROT2x1_MATRIX	/120tx/include/sim_cig_if.h
MSG_ROUND_FIRED	/120tx/include/sim_cig_if.h
MSG_RTN_LT	/120tx/include/sim_cig_if.h
MSG_RTS4x3_MATRIX	/120tx/include/sim_cig_if.h

Data Type	Typedef Location
MSG_SCALE	/120tx/include/sim_cig_if.h
MSG_SHOW_EFFECT	/120tx/include/sim_cig_if.h
MSG_STATICVEH_REM	/120tx/include/sim_cig_if.h
MSG_STATICVEH_STATE	/120tx/include/sim_cig_if.h
MSG_SYS_ERROR	/120tx/include/sim_cig_if.h
MSG_TEST_NAME	/120tx/include/sim_cig_if.h
MSG_TRAJ_CHORD	/120tx/include/sim_cig_if.h
MSG_TRAJ_ENTRY	/120tx/include/sim_cig_if.h
MSG_TRAJ_ENTRY_XFER	/120tx/include/sim_cig_if.h
MSG_TRAJ_TABLE_XFER	/120tx/include/sim_cig_if.h
MSG_TRANSLATION	/120tx/include/sim_cig_if.h
MSG_VIEW_FLAGS	/120tx/include/sim_cig_if.h
MSG_VIEW_MAGNIFICATION	/120tx/include/sim_cig_if.h
MSG_VIEW_MODE	/120tx/include/sim_cig_if.h
MSG_VIEWPORT_STATE	/120tx/include/sim_cig_if h
MSGS_BLK	/120tx/include/sim_cig_if.h
MTXUNION	/120tx/include/sim_cig_if.h
MX_DEVICE	/120/tx/include/mx_defines.h
OMODEL	/120tx/include/structures.h
OVERLAY_PARAMS	/120tx/include/configtree_str.h
POLY_INFO_WORD	/120tx/include/structures.h
POLYGON_LIST	/120tx/include/structures.h
PROJ_DATA	/120tx/include/structures.h
PROJ_DATA_2	/120tx/include/structures.h
R4BBOX2D	/120tx/include/dgi_stdg.h
R4BBOX3D	/120tx/include/dgi_stdg.h
R4HSL	/120tx/include/dgi_stdg.h
R4HSLO	/120tx/include/dgi_stdg.h
R4MTX4X3	/120tx/include/dgi_stdg.h
R4MTX4X4	/120tx/include/dgi_stdg.h
R4P2D	/120tx/include/dgi_stdg.h
R4P3D	/120tx/include/dgi_stdg.h
R4P4D	/120tx/include/dgi_stdg.h
R4RGB	/120tx/include/dgi_stdg.h
R4RGBO	/120tx/include/dgi_stdg.h
R8BBOX2D	/120tx/include/dgi_stdg.h
R8BBOX3D	/120tx/include/dgi_stdg.h
R8HSL	/120tx/include/dgi_stdg.h
R8HSLO	/120tx/include/dgi_stdg.h
R8MTX4X3	/120tx/include/dgi_stdg.h
R8MTX4X4	/120tx/include/dgi_stdg.h
R8P2D	/120tx/include/dgi_stdg.h
R8P3D	/120tx/include/dgi_stdg.h
R8P4D	/120tx/include/dgi_stdg.h
R8RGB	/120tx/include/dgi_stdg.h
R8RGBO	/120tx/include/dgi_stdg.h
RCL_UNION	/120tx/include/rcinclude.h
REAL_4	/120tx/include/dgi_stdc.h
REAL_8	/120tx/include/dgi_stdc.h
RESOLUTION	/120tx/include/sim_cig_if.h
RGBPOLY_LIST	/120tx/include/structures.h
ROOT	/120tx/include/bflydisk.h, /120tx/source/source/find_fn.c
ROT2x1_MTX	/120tx/include/sim_cig_if.h
RTS3x3_MTX	/120tx/include/sim_cig_if.h
RTS4x3_MTX	/120tx/include/sim_cig_if.h
SCREEN	/120tx/include/configtree_str.h
SCRN_CONSTANTS	/120tx/include/configtree_str.h
SEARCH_LIST	/120tx/include/definitions.h

Data Type	Typedef Location
SHOW_EFF	/120tx/include/structures.h
SOMODEL	/120tx/include/structures.h
SREM	/120tx/include/structures.h
STAMP_LIST	/120tx/include/structures.h
STANK	/120tx/include/structures.h
STRING	/120tx/include/dgi_stdc.h
STRUCT2D	/120/tx/include/struct_2d.h
TAC_STATUS	/120tx/include/definitions.h
TANK	/120tx/include/structures.h
tasks	/120tx/include/sysdefs.h, sysdefs2.h
TEXTURE_INDEX	/120tx/include/structures.h
TEXTURE_MAP	/120tx/include/structures.h
TF1	/120tx/include/sim_cig_if.h
TF2	/120tx/include/sim_cig_if.h
TRAJ_DATA	/120tx/include/structures.h
TRAJ_DATA_2	/120tx/include/structures.h
TRAJ_POS	/120tx/include/structures.h
TRAJ_POS_2	/120tx/include/structures.h
UIR4P	/120tx/include/structures.h
UIR4P3D	/120tx/include/structures.h
VIEWPORT_PARAMETERS	/120tx/include/configtree_str.h
VPPOS_ARRAY	/120tx/include/configtree_str.h
WHERE_PROCESS	/120tx/include/ecompiler1.h
WINDOW_DESCRIPTOR_TABLE	/120/tx/include/struct_2d.h
WORD	/120tx/include/dgi_stdc.h

E.3 Function Names To Source File Location

The following list shows each function in the CIG real-time software, and identifies the file in which the function is located. The third column shows the section number in which the function is described in this document.

Function Name	Location	Section
aam_malloc	/120tx/source/config/aam_manager.c	2.2.1.1.1
active_area_init	/120tx/source/source/aa_init.c	2.2.3.1
apinit	/120tx/source/source/rtt.c	2.1.1.1
b0_aam_centroid	/120tx/ballist/source/b0/b0_aam_centroid.c	2.5.2.1
b0_aam_sw_corner	/120tx/ballist/source/b0/b0_aam_sw_corner.c	2.5.2.2
b0_add_static_vehicle	/120tx/ballist/source/b0/b0_add_static_vehicle.c	2.5.2.3
b0_add_traj_table	/120tx/ballist/source/b0/b0_add_traj_table.c	2.5.2.4
b0_bal_config	/120tx/ballist/source/b0/b0_bal_config.c	2.5.2.5
b0_bvol_entry	/120tx/ballist/source/b0/b0_bvol_entry.c	2.5.2.6
b0_cancel_round	/120tx/ballist/source/b0/b0_cancel_round.c	2.5.2.7
b0_cig_frame_rate	/120tx/ballist/source/b0/b0_cig_frame_rate.c	2.5.2.8
b0_database_info	/120tx/ballist/source/b0/b0_database_info.c	2.5.2.9
b0_delete_static_vehicle	/120tx/ballist/source/b0/b0_delete_static_vehicle.c	2.5.2.10
b0_delete_traj_table	/120tx/ballist/source/b0/b0_delete_traj_table.c	2.5.2.11
b0_dummy	/120tx/ballist/source/b0/b0_dummy.c	2.5.2.12
b0_error_detected	/120tx/ballist/source/b0/b0_error_detected.c	2.5.2.13
b0_inapp_message	/120tx/ballist/source/b0/b0_inapp_message.c	2.5.2.14
b0_lm_read	/120tx/ballist/source/b0/b0_lm_read.c	2.5.2.15
b0_model_directory	/120tx/ballist/source/b0/b0_model_directory.c	2.5.2.16
b0_model_entry	/120tx/ballist/source/b0/b0_model_entry.c	2.5.2.17
b0_new_frame	/120tx/ballist/source/b0/b0_new_frame.c	2.5.2.18
b0_print	/120tx/ballist/source/b0/b0_print.c	2.5.2.19
b0_process_chord	/120tx/ballist/source/b0/b0_process_chord.c	2.5.2.20
b0_process_round	/120tx/ballist/source/b0/b0_process_round.c	2.5.2.21
b0_round_fired	/120tx/ballist/source/b0/b0_round_fired.c	2.5.2.22
b0_state_control	/120tx/ballist/source/b0/b0_state_control.c	2.5.2.23
b0_status_request	/120tx/ballist/source/b0/b0_status_request.c	2.5.2.24
b0_traj_chord	/120tx/ballist/source/b0/b0_traj_chord.c	2.5.2.25
b0_traj_entry	/120tx/ballist/source/b0/b0_traj_entry.c	2.5.2.26
b0_undefined_message	/120tx/ballist/source/b0/b0_undefined_message.c	2.5.2.27
bal_get_db_pos	/120tx/source/source/bal_get_db_pos.c	2.4.1
bal_get_lm_grid	/120tx/source/source/bal_get_lm_grid.c	2.4.2
bbnctype	/120tx/source/config/bbnctype.c	2.2.1.2
blank	/120tx/source/gossip/vt100.c	2.6.16.6
bootup_slave133	/120tx/source/source/upstart.c	2.2.3.26.4
bus_error	/120tx/source/source/bus_error.asm	2.2.3.3
bus_error (Butterfly)	/120tx/source/source/support.c	2.2.3.25.4
bus_error_w	/120tx/source/source/support.c	2.2.3.25.5
bx_bvol_int	/120tx/ballist/source/bt/bx_bvol_int.c	2.5.3.1
bx_chord_intersect	/120tx/ballist/source/bt/bx_chord_intersect.c	2.5.3.2
bx_delete_round	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.2
bx_delete_stat_veh	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.10
bx_dist_sq_pt_line	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.11
bx_free_lm_cache	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.6
bx_get_chord_end	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.4
bx_get_db_pos	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.3
bx_get_lb_from_lm	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.8
bx_get_lm_data	/120tx/ballist/source/bt/bx_get_lm_data.c	2.5.3.4

Function Name	Location	Section
bx_get_lm_grid	/120tx/ballist/source/bt/bx_get_lm_grid.c	2.5.3.5
bx_init	/120tx/ballist/source/main/bx_init.c	2.5.1.2
bx_model_int	/120tx/ballist/source/bt/bx_model_int.c	2.5.3.6
bx_new_bvol	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.5
bx_new_poly	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.7
bx_new_round	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.1
bx_new_stat_veh	/120tx/ballist/source/bt/bx_functions.c	2.5.3.3.9
bx_poly_int	/120tx/ballist/source/bt/bx_poly_int.c	2.5.3.7
bx_reset	/120tx/ballist/source/bt/bx_reset.c	2.5.3.8
bx_task	/120tx/ballist/source/main/bx_task.c	2.5.1.3
bx_trajectory	/120tx/ballist/source/bt/bx_trajectory.c	2.5.3.9
cal	/120tx/source/source/cal.c	2.2.3.4
calc_paths	/120tx/source/config/viewport_setup.c	2.2.1.16.2
check_sum	/120tx/source/source/support.c	2.2.3.25.11
cig_2d_setup	/120tx/source/2d/cig_2d_setup.c	2.2.4.2
cig_config	/120tx/source/config/cig_config.c	2.2.1.3.1
compare_buffers	/120tx/force/forcetask.c	2.8.4.2
compile_2d	/120tx/source/2d/cig_comp_2d.c	2.2.4.3
concat_mtx	/120tx/source/config/concat_mtx.c	2.2.1.4
confignode_setup	/120tx/source/config/confignode_setup.c	2.2.1.5
ctoi	/120tx/source/source/support.c	2.2.3.25.14
cup	/120tx/source/gossip/vt100.c	2.6.16.1
data_type	/120tx/force/data_type.c	2.8.1
db_mcc_setup	/120tx/source/source/db_mcc_setup.c	2.2.3.5
dcode_dr11w	/120tx/source/gossip/gossip.c	2.6.15.5
debug_initdr	/120tx/source/source/debug_initdr.c	2.2.3.6
ded_model_trace	/120tx/source/source/ded_model_trace.c	2.2.3.7
display	/120tx/source/gossip/dtp_emu.c	2.6.1.2
display_packet	/120tx/source/gossip/gossip.c	2.6.15.3
double_bot	/120tx/source/gossip/vt100.c	2.6.16.4
double_off	/120tx/source/gossip/vt100.c	2.6.16.5
double_top	/120tx/source/gossip/vt100.c	2.6.16.3
download_bvols	/120tx/source/source/download_bvols.c	2.2.3.8
dr	/120tx/source/source/dr.c	2.2.3.9.1
dr_is_okay	/120tx/source/source/dr.c	2.2.3.9.2
dtp_compiler	/120tx/source/gen_dtp/dtp_compiler.c	2.2.2.1
dtp_emu	/120tx/source/gossip/dtp_emu.c	2.6.1.1
dtp_malloc	/120tx/source/gen_dtp/dtp_funcs.c	2.2.2.2.5
dtp_malloc_init	/120tx/source/gen_dtp/dtp_funcs.c	2.2.2.2.6
dtp_trav1	/120tx/source/gen_dtp/dtp_trav1.c	2.2.2.3
dtp_trav2	/120tx/source/gen_dtp/dtp_trav2.c	2.2.2.4
dynamic_aam_init	/120tx/source/config/aam_manager.c	2.2.1.1.4
excep_init	/120tx/force/exception.asm	2.8.2.1
file_control	/120tx/source/source/file_control.c	2.2.3.10
fill_tree	/120tx/source/config/fill_tree.c	2.2.1.6.1
find_fn	/120tx/source/source/find_fn.c	2.2.3.11
flea	/120tx/source/flea/flea.c	2.7.1
flea_decode_data	/120tx/source/flea/flea_decode_data.c	2.7.2
flea_encode_data	/120tx/source/flea/flea_encode_data.c	2.7.3
flea_init_cig_sw	/120tx/source/flea/flea_init_cig_sw.c	2.7.4
flea_update_pos	/120tx/source/flea/flea_update_pos.c	2.7.5
free133	/120tx/ballist/source/main/slave133_functions.c	2.5.1.4.2
free_configtree	/120tx/source/config/cig_config.c	2.2.1.3.3
ftoh	/120tx/source/gossip/dtp_emu.c	2.6.1.6
fxbvtofl	/120tx/source/source/fxbvtofl.c	2.2.3.12.1
fxbvtofl_020	/120tx/source/source/fxbvtofl.c	2.2.3.12.3
fxbvtofl_dart	/120tx/source/source/fxbvtofl.c	2.2.3.12.2
generic_lm	/120tx/source/source/generic_lm.c	2.3.1.2

Function Name	Location	Section
get_binary_data	/120tx/source/source/support.c	2.2.3.25.12
get_char	/120tx/source/source/support.c	2.2.3.25.13
get_lm	/120tx/source/gossip/dtp_emu.c	2.6.1.9
get_msg_2d	/120tx/source/2d/cig_getm_2d.c	2.2.4.4
get_record	/120tx/source/source/support.c	2.2.3.25.8
get_thing	/120tx/source/2d/get_thing.c	2.2.4.8
getch	/120tx/source/config/getch.c	2.2.1.7
getlmdp	/120tx/source/source/load_modules.c	2.3.2.1
getmatrix	/120tx/source/source/mkmtx_nt.c	2.2.3.19.10
getside	/120tx/source/source/load_modules.c	2.3.2.2
gos_120tx	/120tx/source/gossip/gos_120tx.c	2.6.2
gos_atp	/120tx/source/gossip/gos_atp.c	2.6.3
gos_bal_query	/120tx/source/gossip/gos_bal_query.c	2.6.4
gos_db_query	/120tx/source/gossip/gos_db_query.c	2.6.5.1
gos_display_db_info	/120tx/source/gossip/gos_db_query.c	2.6.5.2
gos_dr11_query	/120tx/source/gossip/gos_dr11_query.c	2.6.6
gos_flea_if	/120tx/source/gossip/gos_flea_if.c	2.6.7
gos_flea_options	/120tx/source/gossip/gos_flea_options.c	2.6.8
gos_fly	/120tx/source/gossip/gos_fly.c	2.6.9
gos_locate	/120tx/source/gossip/gos_locate.c	2.6.10
gos_memory	/120tx/source/gossip/gos_memory.c	2.6.11
gos_model	/120tx/source/gossip/gos_model.c	2.6.12
gos_polys	/120tx/source/gossip/gos_polys.c	2.6.13
gos_single_step	/120tx/source/gossip/gossip.c	2.6.15.6
gos_system	/120tx/source/gossip/gos_system.c	2.6.14
gossip	/120tx/source/gossip/gossip.c	2.6.15.2
gsp_io	/120tx/force/gsp_io.c	2.8.5
gsp_ioctl_read	/120tx/force/force.asm	2.8.3.4
gsp_ioctl_ write	/120tx/force/force.asm	2.8.3.3
gsp_load	/120tx/source/source/gsp_load.c	2.2.3.13
gsp_read	/120tx/force/force.asm	2.8.3.2
gsp_write	/120tx/force/force.asm	2.8.3.1
hexdisplay	/120tx/source/gossip/dtp_emu.c	2.6.1.5
htof	/120tx/source/gossip/dtp_emu.c	2.6.1.7
hw_test	/120tx/source/source/hw_test.c	2.2.3.15
hxflt	/120tx/source/gossip/dtp_emu.c	2.6.1.4
id_4x3mtx	/120tx/source/source/mkmtx_nt.c	2.2.3.19.6
id_matrix	/120tx/source/source/make_bbn.c	2.2.3.17.6
init_configtree	/120tx/source/config/cig_config.c	2.2.1.3.2
init_dtp_stacks	/120tx/source/gen_dtp/dtp_funcs.c	2.2.2.2.4
init_generic_lm	/120tx/source/source/generic_lm.c	2.3.1.1
init_ports	/120tx/force/force.asm	2.8.3.5
init_stuff	/120tx/source/2d/init_stuff.c	2.2.4.9
linkup	/120tx/source/2d/cig_link_2d.c	2.2.4.5
load_dbase	/120tx/source/source/load_dbase.c	2.2.3.16
load_modules	/120tx/source/source/load_modules.c	2.3.2.4
local_terrain	/120tx/source/source/loc_ter.c	2.4.3.2
m1_gun_overlay	/120tx/source/source/gun_overlays.c	2.2.3.14.1
m2_gun_overlay	/120tx/source/source/gun_overlays.c	2.2.3.14.2
main (ballistics)	/120tx/ballist/source/main/bx147_main.c	2.5.1.1
main (force)	/120tx/force/forcetask.c	2.8.4.1
main (gossip)	/120tx/source/gossip/gossip.c	2.6.15.1
main (local_terrain)	/120tx/source/source/loc_ter.c	2.4.3.1
main (rowcol_rd)	/120tx/source/source/rowcol_rd.c	2.3.3.1
main (upstart)	/120tx/source/source/upstart.c	2.2.3.26.1
make_cal_overlay	/120tx/source/source/mkcal.c	2.2.3.18.1
make_m1_overlays	/120tx/source/source/gun_overlays.c	2.2.3.14.3
make_m2_overlays	/120tx/source/source/gun_overlays.c	2.2.3.14.4

Function Name	Location	Section
make_p_nt	/120tx/source/source/mkmtx_nt.c	2.2.3.19.1
mat_mult	/120tx/source/gossip/dtp_emu.c	2.6.1.8
matrix2	/120tx/source/source/mkmtx_nt.c	2.2.3.19.11
model_mtx	/120tx/source/source/model_mtx.c	2.2.3.20
mtxcpy	/120tx/source/source/mkmtx_nt.c	2.2.3.19.12
mult_4x3mtx	/120tx/source/source/mkmtx_nt.c	2.2.3.19.9
multmatrix	/120tx/source/source/make_bbn.c	2.2.3.17.5
mx_error	/120tx/ballist/source/mx/mx_error.c	2.5.4.1
mx_open	/120tx/ballist/source/mx/mx_open.c	2.5.4.2
mx_peek	/120tx/ballist/source/mx/mx_peek.c	2.5.4.3
mx_push	/120tx/ballist/source/mx/mx_push.c	2.5.4.4
mx_skip	/120tx/ballist/source/mx/mx_skip.c	2.5.4.5
mx_wcopy	/120tx/ballist/source/mx/mx_wcopy.c	2.5.4.6
nmi_type	/120tx/force/nmi_type.c	2.8.6
open_dbase	/120tx/source/source/open_dbase.c	2.2.3.21
open_ded	/120tx/source/source/open_ded.c	2.2.3.22
outdisplay	/120tx/source/gossip/dtp_emu.c	2.6.1.3
overlay_setup	/120tx/source/config/overlay_setup.c	2.2.1.9
parser	/120tx/source/config/read_configfile.c	2.2.1.12.6
pix_mult	/120tx/source/source/mkcal.c	2.2.3.18.2
poll_ready	/120tx/force/poll_ready.c	2.8.7
pop_node	/120tx/source/gen_dtp/dtp_funcs.c	2.2.2.2.2
power	/120tx/source/config/fill_tree.c	2.2.1.6.2
process_command	/120tx/source/2d/proc_cmd.c	2.2.4.12
process_vflags	/120tx/source/config/process_vflags.c	2.2.1.10
process_vppos	/120tx/source/config/process_vppos.c	2.2.1.11
prt_mtx	/120tx/source/source/make_bbn.c	2.2.3.17.1
push_node	/120tx/source/gen_dtp/dtp_funcs.c	2.2.2.2.1
qassign	/120tx/source/source/rtt.c	2.1.1.2
r4mat_dump	/120tx/source/config/mat_dump.c	2.2.1.8.1
r4vec_dump	/120tx/source/config/vec_dump.c	2.2.1.15.1
r8mat_dump	/120tx/source/config/mat_dump.c	2.2.1.8.2
r8vec_dump	/120tx/source/config/vec_dump.c	2.2.1.15.2
rcl_command	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.11
rcl_component	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.12
rcl_data	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.13
rcl_init_adrs	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.6
rcl_init_stack	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.1
rcl_lblcmd	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.10
rcl_patch_adrs	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.4
rcl_pop	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.3
rcl_push	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.2
rcl_rtn_adrs	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.7
rcl_set_cntlbl	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.9
rcl_set_errptr	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.5
rcl_set_label	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.8
rcl_stuff_data	/120tx/source/gen_dtp/rcfuncs.c	2.2.2.5.14
read_configfile	/120tx/source/config/read_configfile.c	2.2.1.12.1
read_stuff	/120tx/force/read_stuff.c	2.8.8
read_watch	/120tx/source/source/support.c	2.2.3.25.2
REAL4_fscanf	/120tx/source/config/read_configfile.c	2.2.1.12.4
restore_cur	/120tx/source/gossip/vt100.c	2.6.16.8
return_aam_ptr	/120tx/source/config/aam_manager.c	2.2.1.1.2
rotate_x	/120tx/source/source/make_bbn.c	2.2.3.17.2
rotate_x_nt	/120tx/source/source/mkmtx_nt.c	2.2.3.19.2
rotate_y	/120tx/source/source/make_bbn.c	2.2.3.17.3
rotate_y_nt	/120tx/source/source/mkmtx_nt.c	2.2.3.19.3
rotate_z	/120tx/source/source/make_bbn.c	2.2.3.17.4

Function Name	Location	Section
rotate_z_nt	/120tx/source/source/mkmtx_nt.c	2.2.3.19.4
rowcol_rd	/120tx/source/source/rowcol_rd.c	2.3.3.2
s_step	/120tx/source/gossip/gossip.c	2.6.15.4
save_cur	/120tx/source/gossip/vt100.c	2.6.16.7
scale_mtx	/120tx/source/source/mkmtx_nt.c	2.2.3.19.7
scroll_reg	/120tx/source/gossip/vt100.c	2.6.16.9
send_data	/120tx/source/source/support.c	2.2.3.25.9
setup_bit_blt	/120tx/source/2d/bit_blt.c	2.2.4.1
setup_comp_start	/120tx/source/2d/comp.c	2.2.4.6
setup_define_string	/120tx/source/2d/string.c	2.2.4.13
setup_define_window	/120tx/source/2d/window.c	2.2.4.15
setup_draw_line	/120tx/source/2d/draw_line.c	2.2.4.7
setup_oval_rectangle	/120tx/source/2d/oval_rect.c	2.2.4.10
setup_poly	/120tx/source/2d/poly.c	2.2.4.11
setup_text	/120tx/source/2d/text.c	2.2.4.14
sgr	/120tx/source/gossip/vt100.c	2.6.16.2
simulation	/120tx/source/source/simulation.c	2.2.3.23
slave133_malloc	/120tx/ballist/source/main/slave133_functions.c	2.5.1.4.1
sload	/120tx/source/source/support.c	2.2.3.25.7
spur_int	/120tx/force/exception.asm	2.8.2.2
start_watch	/120tx/source/source/support.c	2.2.3.25
stdio	/120tx/source/source/stdio.c	2.2.3.24
stop_watch	/120tx/source/source/support.c	2.2.3.25.3
STRING_fscanf	/120tx/source/config/read_configfile.c	2.2.1.12.5
string_to_int	/120tx/source/config/read_configfile.c	2.2.1.12.3
swap_axis	/120tx/source/source/mkmtx_nt.c	2.2.3.19.5
sysrup_off	/120tx/source/source/support.c	2.2.3.25.17
sysrup_on	/120tx/source/source/support.c	2.2.3.25.16
system	/120tx/source/source/support.c	2.2.3.25.6
system_aam_init	/120tx/source/config/aam_manager.c	2.2.1.1.3
tassign	/120tx/source/source/rtt.c	2.1.1.3
templates_init	/120tx/source/source/upstart.c	2.2.3.26.2
test_gsp	/120tx/force/test_gsp.c	2.8.9
translate	/120tx/source/source/mkmtx_nt.c	2.2.3.19.8
unbf_getchar	/120tx/source/source/support.c	2.2.3.25.15
update_fov	/120tx/source/config/update_fov.c	2.2.1.13.1
update_rez	/120tx/source/config/update_rez.c	2.2.1.14
upstart	/120tx/source/source/upstart.c	2.2.3.26.3
ver_data	/120tx/source/source/support.c	2.2.3.25.10
viewport_init	/120tx/source/config/viewport_setup.c	2.2.1.16.3
viewport_setup	/120tx/source/config/viewport_setup.c	2.2.1.16.1
viewspace_mtx	/120tx/source/config/update_fov.c	2.2.1.13.2
what_node_on_stack	/120tx/source/gen_dtp/dtp_funcs.c	2.2.2.2.3
whatdirptr	/120tx/source/source/load_modules.c	2.3.2.3
WORD_fscanf	/120tx/source/config/read_configfile.c	2.2.1.12.2

E.4 Macro Names To Source File Location

The following list shows each macro function used by the CIG real-time software, and identifies the file in which the macro is defined. The third column shows the section number in which the macro is described in this document.

Macro Name	Location	Section
AAREAD	/120tx/include/definitions.h	B.1
ABSVAL	/120tx/include/definitions.h	B.2
BCOPY	/120tx/include/mx_defines.h	B.3
CHECK_CLOCK	/120tx/force/force_defines.h	B.4
CHECK_FORCE	/120tx/source/gossip/gos_120tx.c	B.5
DART_ENQUEUE	/120tx/include/functions.h	B.6
DELETE_ROUND	/120tx/include/bx_macros.h	B.7
DELETE_STAT_VEH	/120tx/include/bx_macros.h	B.8
DOWNLOAD_DATA	/120tx/source/2d/cig_link_2d.c	B.9
dtp_bcn	/120tx/include/rcinclude.h	B.10
dtp_bcnr	/120tx/include/rcinclude.h	B.10
dtp_bcnrs	/120tx/include/rcinclude.h	B.10
dtp_bcns	/120tx/include/rcinclude.h	B.10
dtp_bcz	/120tx/include/rcinclude.h	B.10
dtp_bczr	/120tx/include/rcinclude.h	B.10
dtp_bczrs	/120tx/include/rcinclude.h	B.10
dtp_bczs	/120tx/include/rcinclude.h	B.10
dtp_bdgr	/120tx/include/rcinclude.h	B.10
dtp_bdgrs	/120tx/include/rcinclude.h	B.10
dtp_bdlr	/120tx/include/rcinclude.h	B.10
dtp_bdlrs	/120tx/include/rcinclude.h	B.10
dtp_bgn	/120tx/include/rcinclude.h	B.10
dtp_bgns	/120tx/include/rcinclude.h	B.10
dtp_bgz	/120tx/include/rcinclude.h	B.10
dtp_bgzs	/120tx/include/rcinclude.h	B.10
dtp_blm	/120tx/include/rcinclude.h	B.10
dtp_bnz	/120tx/include/rcinclude.h	B.10
dtp_bnzr	/120tx/include/rcinclude.h	B.10
dtp_bnzrs	/120tx/include/rcinclude.h	B.10
dtp_bnzs	/120tx/include/rcinclude.h	B.10
dtp_bpc	/120tx/include/rcinclude.h	B.10
dtp_bpcx	/120tx/include/rcinclude.h	B.10
dtp_bru	/120tx/include/rcinclude.h	B.10
dtp_brur	/120tx/include/rcinclude.h	B.10
dtp_brurs	/120tx/include/rcinclude.h	B.10
dtp_brus	/120tx/include/rcinclude.h	B.10
dtp_brz	/120tx/include/rcinclude.h	B.10
dtp_brzr	/120tx/include/rcinclude.h	B.10
dtp_brzrs	/120tx/include/rcinclude.h	B.10
dtp_brzs	/120tx/include/rcinclude.h	B.10
dtp_dot	/120tx/include/rcinclude.h	B.10
dtp_elm	/120tx/include/rcinclude.h	B.10
dtp_end	/120tx/include/rcinclude.h	B.10
dtp_fov	/120tx/include/rcinclude.h	B.10
dtp_fovr	/120tx/include/rcinclude.h	B.10
dtp_fovrs	/120tx/include/rcinclude.h	B.10
dtp_fovs	/120tx/include/rcinclude.h	B.10
dtp_gdc	/120tx/include/rcinclude.h	B.10

Macro Name	Location	Section
dtp_gdci	/120tx/include/rcinclude.h	B.10
dtp_gdcir	/120tx/include/rcinclude.h	B.10
dtp_gdcirs	/120tx/include/rcinclude.h	B.10
dtp_gdcis	/120tx/include/rcinclude.h	B.10
dtp_gdcn	/120tx/include/rcinclude.h	B.10
dtp_gdcnr	/120tx/include/rcinclude.h	B.10
dtp_gdcnrs	/120tx/include/rcinclude.h	B.10
dtp_gdcns	/120tx/include/rcinclude.h	B.10
dtp_gdcr	/120tx/include/rcinclude.h	B.10
dtp_gdcrs	/120tx/include/rcinclude.h	B.10
dtp_gdcs	/120tx/include/rcinclude.h	B.10
dtp_gr	/120tx/include/rcinclude.h	B.10
dtp_lmi	/120tx/include/rcinclude.h	B.10
dtp_lmir	/120tx/include/rcinclude.h	B.10
dtp_lmirs	/120tx/include/rcinclude.h	B.10
dtp_lmis	/120tx/include/rcinclude.h	B.10
dtp_lod	/120tx/include/rcinclude.h	B.10
dtp_lodr	/120tx/include/rcinclude.h	B.10
dtp_lodrs	/120tx/include/rcinclude.h	B.10
dtp_lods	/120tx/include/rcinclude.h	B.10
dtp_lwd	/120tx/include/rcinclude.h	B.10
dtp_lwdr	/120tx/include/rcinclude.h	B.10
dtp_lwdrs	/120tx/include/rcinclude.h	B.10
dtp_lwds	/120tx/include/rcinclude.h	B.10
dtp_mml	/120tx/include/rcinclude.h	B.10
dtp_mmpre	/120tx/include/rcinclude.h	B.10
dtp_mmpst	/120tx/include/rcinclude.h	B.10
dtp_mwd	/120tx/include/rcinclude.h	B.10
dtp_ngc	/120tx/include/rcinclude.h	B.10
dtp_oio	/120tx/include/rcinclude.h	B.10
dtp_oos	/120tx/include/rcinclude.h	B.10
dtp_osd	/120tx/include/rcinclude.h	B.10
dtp_osds	/120tx/include/rcinclude.h	B.10
dtp_owd	/120tx/include/rcinclude.h	B.10
dtp_owds	/120tx/include/rcinclude.h	B.10
dtp_owdsc	/120tx/include/rcinclude.h	B.10
dtp_owo	/120tx/include/rcinclude.h	B.10
dtp_owr	/120tx/include/rcinclude.h	B.10
dtp_owrs	/120tx/include/rcinclude.h	B.10
dtp_owrsc	/120tx/include/rcinclude.h	B.10
dtp_rc	/120tx/include/rcinclude.h	B.10
dtp_sub	/120tx/include/rcinclude.h	B.10
dtp_subr	/120tx/include/rcinclude.h	B.10
dtp_subrs	/120tx/include/rcinclude.h	B.10
dtp_subs	/120tx/include/rcinclude.h	B.10
dtp_tbc	/120tx/include/rcinclude.h	B.10
dtp_tbdr	/120tx/include/rcinclude.h	B.10
dtp_tbdrs	/120tx/include/rcinclude.h	B.10
dtp_tbrr	/120tx/include/rcinclude.h	B.10
dtp_tbrrs	/120tx/include/rcinclude.h	B.10
DUMP_DART_BUFFER	/120tx/include/functions.h	B.11
ERRMSG	/120tx/source/gen_dtp/rcfuncs.c	B.12
EXCHANGE_DATA	/120tx/include/functions.h	B.13
EXCHANGE_DATA_SIM	/120tx/include/functions.h	B.14
EXCHANGE_FLEA_DATA	/120tx/include/functions.h	B.15
FIND_LM	/120tx/include/functions.h	B.16
FLTOFX	/120tx/include/functions.h	B.17
FREE_LM_CACHE	/120tx/include/bx_macros.h	B.18

Macro Name	Location	Section
FXTO881	/120tx/include/functions.h	B.19
FXTOFL	/120tx/include/functions.h	B.20
GET_CHORD_END	/120tx/include/bx_macros.h	B.21
GET_DB_POS	/120tx/include/bx_macros.h	B.22
GET_LB_FROM_LM	/120tx/include/bx_macros.h	B.23
GLOB	/120tx/include/ememory_map.h, memory_map.h	B.24
INCR_COMPONENT	/120tx/source/gen_dtp/rcfuncs.c	B.25
INIT_MTX	/120tx/include/functions.h	B.26
MALLOC	/120tx/include/bx_defines.h	B.27
NEW_ROUND	/120tx/include/bx_macros.h	B.28
NEW_STAT_VEH	/120tx/include/bx_macros.h	B.29
OPEN_EXCHANGE	/120tx/include/functions.h	B.30
OPEN_FLEA_DATA	/120tx/include/functions.h	B.31
PAGE_FORMAT	/120tx/source/gossip/gos_bal_query.c	B.32
poly_ab	/120tx/include/rcinclude.h	B.33
poly_bvc	/120tx/include/rcinclude.h	B.33
poly_efs	/120tx/include/rcinclude.h	B.33
poly_efsr	/120tx/include/rcinclude.h	B.33
poly_flu	/120tx/include/rcinclude.h	B.33
poly_fsw	/120tx/include/rcinclude.h	B.33
poly_gc	/120tx/include/rcinclude.h	B.33
poly_inf	/120tx/include/rcinclude.h	B.33
poly_lmf	/120tx/include/rcinclude.h	B.33
poly_lsc	/120tx/include/rcinclude.h	B.33
poly_mmf	/120tx/include/rcinclude.h	B.33
poly_pc	/120tx/include/rcinclude.h	B.33
poly_poly	/120tx/include/rcinclude.h	B.33
poly_rm1	/120tx/include/rcinclude.h	B.33
poly_rm2	/120tx/include/rcinclude.h	B.33
poly_rm3	/120tx/include/rcinclude.h	B.33
poly_rm4	/120tx/include/rcinclude.h	B.33
poly_sc	/120tx/include/rcinclude.h	B.33
poly_sci	/120tx/include/rcinclude.h	B.33
poly_sec	/120tx/include/rcinclude.h	B.33
poly_sm1	/120tx/include/rcinclude.h	B.33
poly_sm2	/120tx/include/rcinclude.h	B.33
poly_sm3	/120tx/include/rcinclude.h	B.33
poly_sm4	/120tx/include/rcinclude.h	B.33
poly_stamp	/120tx/include/rcinclude.h	B.33
poly_tog	/120tx/include/rcinclude.h	B.33
poly_vtxe	/120tx/include/rcinclude.h	B.33
poly_vtxl	/120tx/include/rcinclude.h	B.33
PRINTD4	/120tx/source/gossip/gos_memory.c	B.34
PRINTD8	/120tx/source/gossip/gos_memory.c	B.35
PRINTHEX4	/120tx/source/gossip/gos_memory.c	B.36
PRINTHEX8	/120tx/source/gossip/gos_memory.c	B.37
READ_CLOCK	/120tx/force/force_defines.h	B.38
RESTART_CLOCK	/120tx/force/force_defines.h	B.39
ROOM4LABEL	/120tx/source/gen_dtp/rcfuncs.c	B.40
ROOMCHECK	/120tx/source/gen_dtp/rcfuncs.c	B.41
SET_OUT_BITS	/120tx/include/definitions.h	B.42
SET_OUT_M2BITS	/120tx/include/definitions.h	B.43
SYSERR	/120tx/include/functions.h	B.44
TORAD	<multiple files; see section B.45 for list>	B.45
toradians	/120tx/source/source/make_bbn.c	B.46
TRIGGER_FORCE	/120tx/include/functions.h	B.47
WAIT_FORCE	/120tx/include/functions.h	B.48
XCLOSE	/120tx/include/definitions.h	B.49

Macro Name	Location	Section
XLSEEK	/120tx/include/definitions.h	B.50
XOPEN	/120tx/include/definitions.h	B.51
XREAD	/120tx/include/definitions.h	B.52
XWRITE	/120tx/include/definitions.h	B.53

INDEX BY SECTION NUMBER