



Defence Research and
Development Canada

Recherche et développement
pour la défense Canada



Software Documentation for CF-18 ACD

*Daniel Minor
Philip S.E. Farrell*

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

Defence R&D Canada – Toronto

Technical Memorandum

DRDC Toronto TM 2002-026

April 2002

Canada

20020618 173

Software Documentation for CF-18 ACD

Daniel Minor

Philip S. E. Farrell

Defence R&D Canada - Toronto

Technical Memorandum

DRDC Toronto TM 2002-026

April 2002

Author

Philip Farrell

FOR DAN MINOR

Dan Minor

Approved by

G.S. Magee

Lochlan Magee, Ph.D.

Head, Simulation and Modelling for Acquisition, Rehearsal, and Training

Approved for release by

K. M. Sutton

K. M. Sutton

Chair, Document Review and Library Committee

Abstract

Since 1998, the Aircraft Crewstation Demonstrator (ACD) has provided the opportunity for scientists and practitioners to review interface designs in a dynamic setting with the human-in-the-loop. This document serves as a reference for the software developed to support the CF-18 ACD that resides at DRDC Toronto.

The ACD has a very modular architecture, which allows for components to be added and removed over time. As such, the ACD is best described in terms of its individual components. The modular nature of the ACD, combined with the physical separation of the software components across several computers, makes interprocess communication of central importance to the software architecture. As such this document gives a comprehensive view of the interprocess communication that occurs during the running of the ACD, before treating each component in depth.

This document is current as of January 17, 2002. It is anticipated that as other components are added to the CF-18 ACD, this reference document will also need to be updated.

Résumé

Depuis 1998, le démonstrateur de poste d'équipage navigant (ACD) a donné l'occasion aux chercheurs et aux spécialistes de revoir la conception des interfaces dans un environnement dynamique auquel est intégré l'être humain. Le présent document sert de référence au logiciel mis au point pour soutenir le démonstrateur de poste d'équipage navigant du CF-18 qui se trouve au RDDC Toronto.

Le démonstrateur est d'une architecture très modulaire en ce qu'il permet d'ajouter ou de retirer des composants avec le temps. Comme tel, le démonstrateur se décrit mieux en fonction de chacun de ses composants. Le caractère modulaire du démonstrateur, combiné à la répartition matérielle des composants du logiciel parmi plusieurs ordinateurs, fait de la communication interprocessus un élément d'une importance primordiale dans l'architecture du logiciel. Comme tel, le présent document donne une vue d'ensemble complète de la communication interprocessus qui se produit pendant le fonctionnement du démonstrateur, avant de traiter de chaque composant en profondeur.

Le présent document est à jour au 17 janvier 2002. Il est prévu qu'à mesure que d'autres composants seront ajoutés au démonstrateur de poste d'équipage navigant du CF-18, le présent document de référence soit également mis à jour.

This page intentionally left blank.

Executive summary

Since 1998, the Aircraft Crewstation Demonstrator (ACD) has provided the opportunity for scientists and practitioners to review interface designs in a dynamic setting with the human-in-the-loop. The ACD continued to be developed with the addition of hardware and software components. There exist three or four versions of the original ACD, one of which is the CF-18 ACD that resides at DRDC Toronto.

This reference document details the software architecture of the CF-18 Aircraft Crewstation Demonstrator. It provides a complete description of the interprocess communication framework for the simulator. The framework's software components are:

- Controls
- Data Collection
- FLSIM
- Scenario Framework
- Helmet Mounted Display
- Message Router
- Polhemus Motion Tracker client
- Simulations.

The description begins with an overview of the software component. Each component is described in terms of its objects and functions. Its relationship to the overall architecture is discussed.

This document is current as of January 17, 2002. It is anticipated that as other components are added to the CF-18 ACD, this reference document will also need to be updated.

Minor, D L., and P.S.E. Farrell 2002. Software Documentation for CF-18 ACD. DRDC Toronto TM 2002-015 Defence R&D Canada Toronto.

Sommaire

Depuis 1998, le démonstrateur de poste d'équipage navigant (ACD) a donné l'occasion aux chercheurs et aux spécialistes de revoir la conception des interfaces dans un environnement dynamique auquel est intégré l'être humain. Le démonstrateur continue d'être développé par l'ajout de composants matériels et logiciels. Il existe trois ou quatre versions du démonstrateur d'origine, dont l'un est le démonstrateur du CF-18, qui se trouve au RDDC Toronto.

Le présent document de référence donne le détail de l'architecture logicielle du démonstrateur de poste d'équipage navigant du CF-18. Il donne une description complète du cadre de communication interprocessus pour le simulateur. Les composants logiciels sont :

- les commandes;
- la saisie des données;
- FLSIM
- le cadre de travail des scénarios;
- l'affichage de casque;
- le routeur de messages;
- le dispositif de suivi de mouvement Polhemus;
- les simulations.

La description commence par une vue d'ensemble du composant logiciel. Chaque composant est décrit en fonction de ses objets et de ses fonctions. Il est question de ses rapports avec l'ensemble de l'architecture.

Le présent document est à jour au 17 janvier 2002. Il est prévu qu'à mesure que d'autres composants seront ajoutés au démonstrateur de poste d'équipage navigant du CF-18, le présent document de référence soit également mis à jour.

Minor, D L ., and P.S.E. Farrell. Software Documentation for CF-18 ACD. DRDC Toronto TM 2002-015 Defence R&D Canada Toronto.

Table of contents

Abstract	i
Résumé	i
Executive summary	iii
Sommaire	iv
Table of contents	v
List of figures	viii
List of tables	ix
Introduction	1
The CF-18 ACD Software Architecture	3
Overview	3
Commercial Off The Shelf (COTS) Products	5
Interprocess Communication	5
Router Communication	5
Direct UDP Communication	6
Shared Memory Communication	6
Summary of Module IPC	7
The Common Library	9
Overview	9
UDPChannel_c	9
Timer_c	10
List_c	10
PeriodicProcess_c	11
Semaphore_c	12
SharedPool_c	13

Controls	15
Overview	15
The Stick and Pedals (cf18_stick)	16
The Throttle (cf18_throttle)	17
The Gear Panel (cf18_gearp)	19
Data Collection.....	21
Overview	21
Database Design.....	21
The Data Collection Program (Datacol).....	23
FLSIM	27
Overview	27
Simulator Application	27
Control Application.....	27
Framework	29
Overview	29
The MessageQueue_c Class.....	29
The Entity_c Class	29
The StaticTarget_c Class.....	30
The StaticThreat_c Class.....	30
The EntityList_c Class	31
Scenario Configuration Files.....	32
The Framework Program	33
Framework Message Types.....	34
Weapons Models.....	35
Helmet Mounted Display	37
Overview	37
SIMEYE XL-100	37
The Scene	37
Global Data Structures	37
Head Position and the Polhemus Tracker Client.....	38
System Pre-Sync Callback	39

FLIR Simulation.....	39
Target Designation	43
Height Above Terrain (HAT) Calculation	43
The Entity List.....	43
The Router.....	45
Overview	45
The Router Server	45
The Router Client.....	46
Source code for a Minimal Client Application	47
The Tracker	49
Overview	49
tracker_c.....	49
The Client Application.....	50
The Sims.....	51
Overview	51
Mission Computer.....	51
Stores Management.....	53
Radar	54
Conclusion.....	57
Annex A	59
Header Files	59
Bibliography.....	73
List of symbols/abbreviations/acronyms/initialisms	75

List of figures

Figure 1. CF-18 ACD Interprocess Communication.....	4
Figure 2. Location of Controls on the Stick	17
Figure 3. Location of Controls on the Throttle	18
Figure 4. Entity Relationship Diagram.....	22
Figure 5. FLIR Symbology Elements.....	41
Figure 6. Stations on a CF-18.....	53

List of tables

Table 1. Summary of Router Message Types	6
Table 2. Summary Direct UDP Communication.....	6
Table 3. Summary of Module IPC	7
Table 4: UDPChannel_c Methods.....	9
Table 5: Timer_c Methods	10
Table 6: List_c Methods.....	11
Table 7: PeriodicProcess_c Methods	12
Table 8: Semaphore_c Methods.....	12
Table 9: SharedPool_c Methods	13
Table 10: Discrete Values produced by the Stick	16
Table 11: Discrete Values produced by the Throttle.....	18
Table 12: Discrete Values produced by the Gear Panel	19
Table 13: Relations.....	23
Table 14:MessageQueue_c Methods.....	29
Table 15: StaticTarget_c Methods	30
Table 16: StaticThreat_c Methods	31
Table 17: EntityList_c Methods.....	32
Table 18: HMD Interprocess Communication	39
Table 19: FLIR Symbology Elements.....	42

Table 20: Router Client Methods.....	47
Table 21: tracker_c Methods.....	49
Table 22: Tracker Data Record Format.....	50
Table 23: Sim Classes	51
Table 24: Mission Computer Discrete Value Handling	52

Introduction

Virtual prototyping is a powerful tool for human factors engineering. The advantage of virtual prototyping is the ability to rapidly visualize the results of design decisions in interface design. When coupled with human-in-the-loop simulation, rapid prototyping allows an emerging design to be evaluated under dynamic conditions. The recent availability of low cost capable computer generated graphics has made this technology accessible to both scientists and practitioners alike.

DRDC Toronto had been developing tools for human factors engineering since 1985. These tools included constructive modelling and simulation environments for design and rapid prototyping. By the mid 1990s the maturity of computer graphics (e.g., 30Hz update rates at an affordable price) made the extension of these tools into a low fidelity, rapidly re-configurable, human-in-the-loop aircraft crewstation simulation feasible. This device was called the Aircraft Crewstation Demonstrator (ACD).

The first ACD was conceived in 1993 and delivered to the Director Technical Airworthiness (DTA) in 1998. A second ACD, for research purposes, was installed at DRDC Toronto in the same year. The idea for the use of the simulators was that new interfaces would be studied with DRDC Toronto's ACD then passed onto DTA's ACD for test and evaluation. If warranted, the interface could be tested in a higher fidelity simulator and then field-tested. The interfaces that have been studied using the ACD include Direct Voice Input for a helicopter's Control Display Unit, a helicopter's Heads Up Display symbology, and comparing different Helmet Mounted Display systems for the CF-18 ACD (which is the current study).

The ACD is a low fidelity, rapidly re-configurable, human-in-the-loop simulator that allows new (virtual or real) equipment or procedures to be attached and tested. The human interface includes a projection of the Out-The-Window (OTW) scene, aircraft instrument displays on computer screens, and aircraft controls. Normally, the ACD is enclosed in a structure that has the look of a cockpit. At present, two shell types exist: 1) medium lift helicopter and 2) fast jet. The ACD with the fast jet shell we call the CF-18 ACD.

The intent of this document is to archive the software architecture of the CF-18 ACD that has been developed over the years by contractors (primarily CMC Electronics and The HFE Group) and research assistants (mostly student computer programmers). This is an opportunity to describe the ACD software configuration in a systematic way for reference purposes. Because capability will be added to the ACD continually, the document is dated and will need to be revised as significant changes take place. It is hoped that scientists, contractors, research assistants, and programmers will find this document useful in reconfiguring the ACD for future studies and tests.

This page intentionally left blank

The CF-18 ACD Software Architecture

Overview

The CF-18 ACD has a very modular design. This has several advantages over a more tightly integrated system. The various components of the simulation can be spread across several computers, allowing for less computing intensive tasks to be done on older computers, thus minimising the cost of the simulator. It also minimizes the interdependencies of the simulator components, allowing for systems to be run in isolation. This in turn contributes to the adaptability and portability of the software, making it relatively easy for the simulator to be modified for various experimental conditions.

Much of this modularity comes from the use of a central message router for interprocess communication, rather than direct connections between the different modules. Each module has to know only the location of the router, not the location of the source of the information it desires. If another module is added that needs access to this information, the first module does not have to be modified; the new module can obtain the information from the message router. Finally, a module can be removed and replaced with another module without any changes being made to the rest of the system. For example, the current architecture makes use of a simple Scenario Framework application that generates and moves computer-controlled entities in the simulation. Should future experiments have more sophisticated requirements, this Framework could be removed and replaced with one or more applications (possibly commercial products) without any changes being made to the rest of the architecture.

A brief overview of each module is given below. Generally, a module corresponds to one subdirectory of the source code, and usually with one compiled application. The modules are discussed in alphabetical order in this document.

- **Controls:** The controls module consists of three applications: `cf18_stick`, `cf18_throttle` and `cf18_gearp`. These applications retrieve information from the controls via BG Cereal Boxes, and communicate the information to the Sims and FLSIM (see below).
- **Datacol:** This module is responsible for data collection for the CF-18 ACD. It subscribes to messages of interest via the router, and stores experimental data in text files.
- **DDIs:** DDI stands for Direct Digital Interface. The DDIs are VAPS applications that display mission information on to the screens of the ACD.
- **FLIR:** The FLIR module provides a visual simulation of a FLIR (Forward Looking Infrared) sensor.
- **FLSIM:** FLSIM is a commercial flight simulator product that provides the flight model for the ACD. It consists of two applications, modified with custom code: `cf18_flsim_sim` runs the flight model, and `cf18_flsim_con` provides a command line utility to control the execution of the flight model.

- **Framework:** As mentioned above, the Scenario Framework provides a simple means of generated targets and threats in the simulation. The scenario is configured via text files.
- **Helmet Mounted Display:** The helmet mounted display provides the visual display of the CF-18 ACD.
- **Router:** The router handles interprocess communication between the different modules of the simulation.
- **Simulations (the Sims):** The Sims are a collection of classes that simulate various flight systems. The majority simply extract data retrieved from FLSIM, and send it to the DDIs to be displayed. Of importance are the Stores Management System, which handles onboard stores and weapons firing, the radar, which provides a simulation of A/A radar and the Mission Computer, which processes control inputs.
- **Tracker:** The tracker module provides a client application for a Polhemus motion tracker.

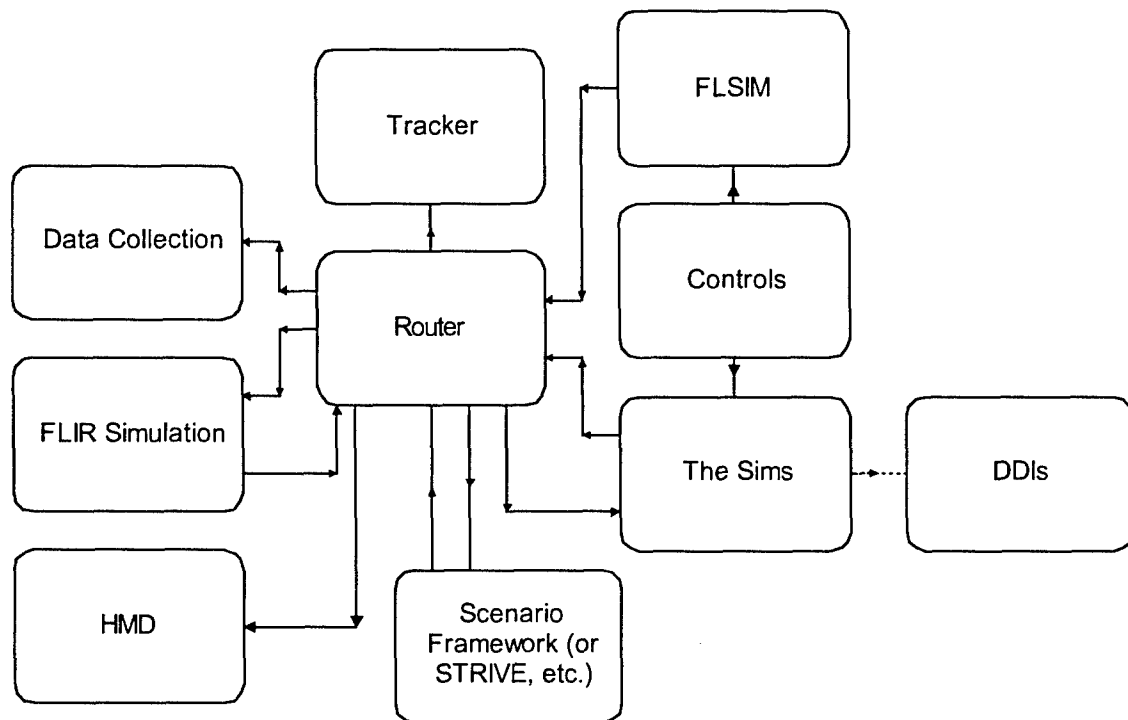


Figure 1. CF-18 ACD Interprocess Communication

In addition to these modules, there are several classes that are used throughout the simulation, for example, the class used for router connections, and a class handling timers. These are described below in the section on Utility classes.

Commercial Off The Shelf (COTS) Products

The CF-18 takes advantage of several commercial software products. These are summarized below:

- **FLSIM:** FLSIM is a commercial flight simulator product that provides a flight model for the CF-18 ACD. FLSIM is a product of Virtual Prototypes Inc.
- **VAPS:** VAPS is a rapid prototyping application for aircraft instrumentation. It is used to draw the DDI displays. VAPS is a product of Virtual Prototypes Inc.
- **VEGA:** VEGA is a 3D graphics API that provides a convenient interface to the SGI OpenGL and Performer APIs. It comes with several modules; one of these, SensorWorks is used to simulate the FLIR sensor. VEGA is a product of Multigen-Paradigm Inc.

Interprocess Communication

As mentioned above, the CF-18 ACD has a very modular design. This makes it quite dependent upon interprocess communication. Much of this communication takes place via the message router, but shared memory and direct socket connections are also used.

All socket communication, including communication via the message router, utilizes the User Datagram Protocol (UDP). UDP was chosen because it is a stateless protocol, which means that the processes communicating with each other do not negotiate a connection, but send data to one another directly. If one process is absent, it will not adversely affect another process, unless that process is directly dependent upon information from the first. This enhances the modularity of the ACD. Also, a stateless protocol avoids the overhead involved in maintaining a connection between two processes, and so is more efficient.

Communication by direct UDP connections and via shared memory is more efficient than communication through the router, but information passed by these mechanisms is inaccessible to data collection. Additionally, it reduces the modularity of the ACD; in particular, communication by shared memory is only possible between processes residing on the same computer. For these reasons, data is passed through the router whenever it is practical to do so.

Router Communication

The router serves as an intermediary between modules wishing to communicate with each other. Currently, there are 17 message types passed through the router. However, the functionality of the router is not dependent upon knowledge of the message types that it is handling. The message types are listed in table 1:

Table 1. Summary of Router Message Types

MESSAGE TYPE	PURPOSE	DATA TYPE
CREATE_ENTITY	Entity creation event.	CreationMsg_s
DESIGNATE	Hold location that the pilot has designated.	DesignateMsg_s
DESTROY_ENTITY	Remove an entity from the simulation.	DestroyMsg_s
EXPLODE_ENTITY	Cause an entity to explode.	ExplodeEntityMsg_s
FLIR_ACTIVE	Indicate that the FLIR is active.	None.
FLIR_DATA	Hold status of FLIR controls.	flir_data_s
HAT	Indicate height above terrain.	HatMsg_s
MASTER_MODE	Indicates a change in the master mode of the CF-18.	MasterModeMsg_s
POSITION_ENTITY	Position an entity.	PosUpdateMsg_s
POSITION_OWNERSHIP	Hold positional information for the CF-18.	PosUpdateMsg_s
RADAR_UPDATE	Target information for the radar simulation.	RadarMsg_s
THREAT_DETECTED	Indicate a threat has been detected.	None.
TRACKER_UPDATE	Hold status of Polhemus Tracker.	PosData_s
TT_REL	Time until weapon can be released.	TTRelMsg_s
TTI	Time until weapon impact with target	TTIMsg_s
UNDESIGNATE	Release designation.	None.
WEAPON_FIRED	Hold weapon fire information.	WeaponFireMsg_s

Direct UDP Communication

Some components of the simulation bypass the message router and use direct UDP connections in order to send information. This is more efficient than using the router; however, the information sent is not visible to the data collection utility, and the overall modularity of the ACD is reduced. Direct UDP connections are summarized in Table 2:

Table 2. Summary Direct UDP Communication

MESSAGE TYPE	MESSAGE SUBTYPE	DATA TYPE
CF18_MSG_FLT_CTRL	CF18_FLTCTL_STREAM_STICK_DATA	CF18_FltCtl_StickPedalGearP_s
	CF18_FLTCTL_STREAM_THR_DATA	CF18_FltCtl_Throttle_s
	CF18_FLTCTL_STREAM_TDC_DATA	CF18_FltCtl_TDC_s
	CF18_FLTCTL_EVENT_DATA	CF18_FltCtl_Discrete_s
CF18_MSG_DISPLAY	N/A	CF18_MessageToDisplays_s
CF18_MSG_AUDIO	N/A	CF18_Audio_s
CF18_FLSIM_EXPORT	N/A	Flsim_Export_Msg

Configuration for the UDP connections is done by editing the udp_settings.h file found in the common directory on ACD2. It is necessary to recompile the affected applications in order for the changes to take effect.

Shared Memory Communication

Communication between the Sims and the DDIs takes place via shared memory. Shared memory allows one process direct access to memory belonging to another process, and hence provides a quick, efficient mechanism for interprocess communication.

Summary of Module IPC

This table presents a list of the modules in the CF-18 ACD that communicate via the router, and details the messages types that each module sends and receives.

Table 3. Summary of Module IPC

MODULE	SENDS	RECEIVES
Controls	CF18_FLTCLT_STREAM_TDC_DATA CF18_FLTCTL_EVENT_DATA CF18_FLTCTL_STREAM_STICK_DATA CF18_FLTCTL_STREAM_THR_DATA	None.
Datacol	None.	CREATE_ENTITY DESTROY_ENTITY EXPLODE_ENTITY FLIR_ACTIVE POSITION_ENTITY POSITION_OWNERSHIP THREAT_DETECTED TRACKER_UPDATE WEAPON_FIRED
DDI's		CF18_MSG_DISPLAY
FLIR Simulation	DESIGNATE UNDESIGNATE	CREATE_ENTITY DESTROY_ENTITY EXPLODE_ENTITY FLIR_DATA POSITION_ENTITY TT_REL TTI
FLSIM	CF18_FLSIM_EXPORT POSITION_OWNERSHIP	CF18_FLTCTL_STREAM_STICK_DATA CF18_FLTCTL_STREAM_THR_DATA HAT
Framework	CREATE_ENTITY DESTROY_ENTITY EXPLODE_ENTITY POSITION_ENTITY	POSITION_OWNERSHIP TRACKER_UPDATE TTI WEAPON_FIRED
HMD	HAT	CREATE_ENTITY DESTROY_ENTITY EXPLODE_ENTITY POSITION_ENTITY POSITION_OWNERSHIP TT_REL TTI
The Sims	CF18_MSG_DISPLAY FLIR_ACTIVE MASTER_MODE THREAT_DETECTED TT_REL TTI WEAPON_FIRED	CF18_FLSIM_EXPORT CF18_FLTCLT_STREAM_TDC_DATA CF18_FLTCTL_EVENT_DATA CF18_MSG_AUDIO DESIGNATE RADAR_UPDATE UNDESIGNATE
Tracker	TRACKER_UPDATE	None.

This page intentionally left blank

The Common Library

Overview

A statically linked archive, called libcommon.a, contains several classes used throughout the CF-18 ACD source code. Having the code contained within a common library avoids the administrative problems that result from multiple versions of the same code spread throughout the source tree and ensures that updates made to the code are reflected in every application that makes use of it.

The header files for the components of the common library appear in Annex A, for ease of reference.

UDPChannel_c

UDPChannel_c is used for all UDP communications in the CF-18 ACD, including those involving the router. It is defined in the files udp_comm.h and udp_comm.cpp. Its interface is defined by the following methods:

Table 4: UDPChannel_c Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
UDPChannel_c	UDP_SEND, or UDP_RECEIVE, or UDP_TRANSCEIVE (default)	Constructor.	None.
~UDPChannel_c	None.	Destructor. Calls Stop, then exits.	None.
Start	None.	Starts the UDP channel.	None.
Stop	None.	Stops the UDP channel.	None.
Set	One of: UDP_RECEIVE_PORT, port no. UDP_SEND_PORT, port no. UDP_RECV_BUF, buffer size UDP_SEND_BUF, buffer size UDP_RECV_ADDR, ip address UDP_SEND_ADDR, ip address	Sets socket properties.	None.
RecvMessage	buffer message size timeout (optional, in milliseconds)	Attempts to receive a message through the socket.	Size of message received, or 0 if no message is received.
SendMessage	buffer message size	Sends a message through the socket.	The value 1, indicating success.

Timer_c

Timer_c provides a convenient wrapper function for the Unix time functions `gettimeofday` and `nanosleep`. It also maintains two variables, `startSeconds`, and `startUSeconds`, which track the time that the Timer_c class object was instantiated, or the time of the last call to `Reset`. This allows for events to be timed. Timer_c is defined in the files `timer.h` and `timer.cpp`.

Table 5: Timer_c Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
Timer_c	None.	Constructor.	None.
~Timer_c	None.	Destructor.	None.
ElapsedMicroseconds	None.	Returns number of microseconds since last reset.	Integer value representing microseconds.
ElapsedMilliseconds	None.	Returns number of milliseconds since last reset.	Integer value representing milliseconds.
ElapsedSeconds	None.	Returns number of seconds since last reset.	Integer value representing seconds.
GetCurrentTimeInSeconds	None.	Returns number of seconds since January 1, 1970.	Integer value representing seconds.
WaitMilliseconds	milliseconds	Waits for a specified number of milliseconds.	None.
Reset	None.	Sets values for <code>startSeconds</code> and <code>startUSeconds</code> to current time.	None.

A common use of Timer_c is to ensure that a given section of code is executed no more often than a given frequency. For example, to ensure that the body of a loop is executed no more frequently than every 10 seconds, the following code would be used:

```
while(!done) {
    timer->Reset();

    ...

    timer->WaitMilliseconds(10 - timer->ElapsedMilliseconds());
}
```

List_c

List_c maintains a linked list made up of nodes with the following structure, which is defined in `list.h`:

```
struct list_node {
    struct list_node *next, *prev;
    int priority;
    void *data;
};
typedef struct list_node ListNode_s;
```

Two functions are provided to maintain nodes: CreateNode and MakeNode. Both functions accept a pointer to a buffer and an integer representing the size of the buffer. CreateNode makes a copy of the contents of the buffer, and has data point to the copy, while MakeNode simply has data point to the location of the buffer. List_c is defined in the files list.h and list.cpp.

The methods defined in List_c are as follows:

Table 6: List_c Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
List_c	None.	Constructor.	None.
~List_c	None.	Destructor.	None.
IsEmpty	None.	Determines whether list is empty or not.	True (1) if list is empty.
Empty	None.	Removes all nodes from list.	None.
FirstElement	None.	Returns first node in list.	First node in list.
LastElement	None.	Returns last node in list.	Last node in list.
NextElement	Pointer to current node.	Returns next node in list.	Next node in list.
PrevElement	Pointer to current node.	Returns previous node in list.	Previous node in list.
InsertAfter	pointer to node. pointer to node to insert.	Inserts a node in to the list.	None.
InsertBefore	pointer to node. pointer to node to insert.	Inserts a node in to the list.	None.
Enqueue	pointer to node, <i>or</i> pointer to buffer size of buffer	Inserts a node at the end of the list.	Node.
AddNodeWithPriority	pointer to node priority, <i>or</i> , pointer to buffer buffer size priority	Inserts a node in to the list by order of priority.	None.
Push	pointer to node	Inserts a node at the beginning of the list.	None.
Delete	pointer to node	Calls free(data), and removes node from list.	None.
Count	None.	Returns number of items in list.	Number of items in list.

PeriodicProcess_c

The PeriodicProcess_c allows a user function to be run at a set period, given in milliseconds.

PeriodicProcess_c starts a thread to run the PeriodicUpdate function. This function utilizes the Timer_c class discussed above to time the execution of the user function. If the user function takes less time than the period of the periodic process, the WaitMilliseconds method of the Timer_c class is called to make up the difference. The user function is not pre-empted if it takes more time than is allotted to it, so this class does not provide true real time functionality. It is possible to pause and resume the user function because the PeriodicUpdate

function waits on a mutex, called `_pauseResume`, at the beginning of each iteration of its main loop.

`PeriodicProcess_c` is implemented in the files `periodic.h` and `periodic.cpp`. The following methods are defined:

Table 7: `PeriodicProcess_c` Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
<code>PeriodicProcess_c</code>	pointer to function period (<i>in milliseconds</i>) 1 = start immediately (<i>default</i>) or 0 = deferred execution	Constructor.	None.
<code>~PeriodicProcess_c</code>	None.	Destructor.	None.
<code>Start</code>	None.	Creates a thread to execute the <code>PeriodicUpdate</code> function.	None.
<code>Pause</code>	None.	Sets <code>_pauseRelease</code> mutex.	None.
<code>Resume</code>	None.	Releases <code>_pauseRelease</code> mutex.	None.
<code>Kill / Exit</code>	None.	Ends execution of <code>PeriodicUpdate</code> function.	None.
<code>SetPeriod</code>	period (<i>in milliseconds</i>)	Sets period.	None.
<code>GetPeriod</code>	None.	Returns the period of the process.	Period in milliseconds.
<code>WaitForCompletion</code>	None	Waits for the thread to terminate.	None.
<code>DoOnce</code>	pointer to function	Calls the <code>pthread_once</code> function, with the user function as an argument.	None.

Semaphore_c

Semaphores provide a means of coordinating data access between two or more processes. In some cases, if two or more processes are allowed to access the same data structure concurrently, unpredictable and undesirable results will occur. Semaphores are used to control concurrent access. Each semaphore has a count associated with it. When the count is zero, a process increments the count, accesses the data structure, then decrements the count. If the count is not zero, a process waits on the semaphore until the count is zero.

The `Semaphore_c` class provides a wrapper around the Unix `semop` system call. The methods are defined as follows:

Table 8: `Semaphore_c` Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
<code>Semaphore_c</code>	Semaphore Id.	Constructor.	None.
<code>~Semaphore_c</code>	None.	Destructor.	None.
<code>Lock</code>	None.	Lock the semaphore.	None.
<code>Unlock</code>	None.	Unlock the semaphore.	None.
<code>Increment</code>	None.	Increments the count on the	None.

		semaphore.	
Decrement	None.	Decrements the count on the semaphore.	None.
Destroy	None.	Destroys the semaphore.	None.

SharedPool_c

SharedPool_c is the shared memory class used in the CF-18 ACD. The SharedPool_c allocates an area of shared memory, and then uses a directory to provide named access to areas of the shared memory pool. Semaphores (see Semaphore_c above) are used to prevent concurrent access to the directory data structure, and so ensure its integrity.

The methods defined by SharedPool_c are detailed below:

Table 9: SharedPool_c Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
SharedPool_c	shared memory key, or, key size number of entries semaphore key	Creates a shared memory pool and attaches to it.	None.
~SharedPool_c	None.	Deletes the semaphore for the memory pool, and then detaches from it.	None.
Destroy	None.	Removes shared memory pool.	
Exists	None.	Returns true if a handle to a shared memory area exists.	True if shared
Allocate	name Name to be used for the area, amount of memory to allocate.	Allocates an area of memory in the shared memory pool.	Pointer to memory area allocated.
Deallocate	Name of the area to deallocated.	Removes a named area of memory from the shared memory pool.	Null.
GetAddressOf	Name of the area for which to get the address.	Returns address of a named area in the shared memory pool.	Pointer to named area in the shared memory pool.
GetSizeOf	Name of the area for which to get the size.	Returns size of a named area in the shared memory pool.	Size in bytes.
Size	None.	Returns the size of the shared memory	Size in bytes.
Lock	None.	Locks shared memory so that updates to the directory structure can take place.	None.
Unlock	None.	Unlocks shared memory.	None.
GetDirectoryHeader	None.	Returns header structure for the shared pool directory.	Pointer to a DirectoryHeader_s.
Directory	Index of the directory header to return	Returns a directory entry from the shared pool directory.	Pointer to a DirectoryEntry_s.

This page intentionally left blank

Controls

Overview

There are three modules responsible for the CF-18 ACD controls. They are `cf18_stick`, `cf18_throttle`, and `cf18_gearp`. They function in a similar way. Each accepts serial input from a Cereal Box, processes it, and sends messages via UDP to the Mission Computer simulation and FLSIM indicating the state of the controls.

The following controls are operational on the ACD:

1. **Stick** (`cf18_stick`)
 - Trigger
 - Undesignate
 - Weapon Select
 - Castle Switch
 - Weapon Release
 - Trim Switch
2. **Pedals** (`cf18_stick`)
3. **Throttle** (`cf18_throttle`)
 - Sensor Switch
 - ATC Switch
 - TDC Position (streaming)
 - TDC (press to designate)
 - Chaff/Flares Switch
 - Comm Switch
 - Cage/Uncage Switch
 - Speed Brake
4. **Gear Panel** (`cf18_gearp`)
 - Gear Switch
 - Flap Switch
 - Parking Brake / Emergency Brake

Note: Although some of the code in the Sims seems to indicate that the bezel switches are operational, this is not the case.

The Stick and Pedals (cf18_stick)

The stick module is responsible for the stick and the pedals. It communicates with the Mission Computer simulation and FLSIM via UDP connections. In addition to stick and pedal position, which are streaming values, the stick produces the following discrete values, which are defined in the cf18_msg_types.h header file, which is included in Annex A.

Table 10: Discrete Values produced by the Stick

CONTROL	DISCRETE VALUES
Trim Switch	CF18_TRIM_SW_LWD CF18_TRIM_SW_RWD CF18_TRIM_SW_AND F18_TRIM_SW_ANU F18_TRIM_SW_OFF
Castle Switch	CF18_SENSORSW_UP CF18_SENSORSW_LFT CF18_SENSORSW_DN CF18_SENSORSW_RGT
Weapon Release	CF18_WPNRELSW_OFF CF18_WPNRELSW_DEP
Weapon Select	CF18_WPNSELSW_GUN CF18_WPNSELSW_AIM9 CF18_WPNSELSW_AIM7
Undesignate	CF18_UNDESIGNSW_OFF CF18_UNDESIGNSW_DEP
Trigger	CF18_TRIGERSW_OFF CF18_TRIGERSW_DET1 CF18_TRIGERSW_DET2
Pedals	CF18_LEFTBRAKE_OFF CF18_LEFTBRAKE_ON CF18_RIGHTBRAKE_OFF CF18_RIGHTBRAKE_ON.

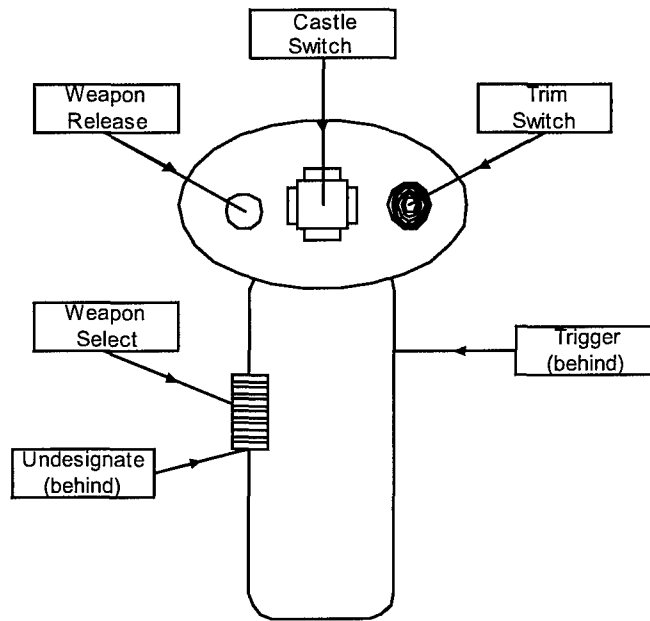


Figure 2. Location of Controls on the Stick

The Throttle (cf18_throttle)

The throttle module is quite similar in function to the stick. It initialises three UDP connections, one to send streaming throttle positional information to FLSIM, one to send TDC positional information to the Mission Computer simulation, and one to send discrete values to the Mission Computer. The discrete values are enumerated in `cf18_msg_types.h` (included in Annex A) and are summarized in the following table:

Table 11: Discrete Values produced by the Throttle

CONTROL	DISCRETE VALUES
Sensor Switch	CF18_RAIDFLIRSW_DEP CF18_RAIDFLIRSW_OFF
ATC Switch	CF18_ATDSW_DEP CF18_ATCSW_OFF
TDC Button	CF18_TDCSW_DEP CF18_TDCSW_OFF
Light Switch	Non-functional.
Comms Switch	CF18_COMMSW_NO1 CF18_COMMSW_NO2 CF18_COMMSW_OFF
Chaff/Flare Switch	CF18_CHAFFLARSW_FLARE CF18_CHAFFLARSW_CHAFF CF18_CHAFFLARSW_OFF
Cage/Uncage Switch	CF18_CAGEUNCAGE_DEP CF18_CAGEUNCAGE_OFF
Speed Brake	CF18_SPEEDBRAKE_RET CF18_SPEEDBRAKE_EXT CF18_SPEEDBRAKE_OFF

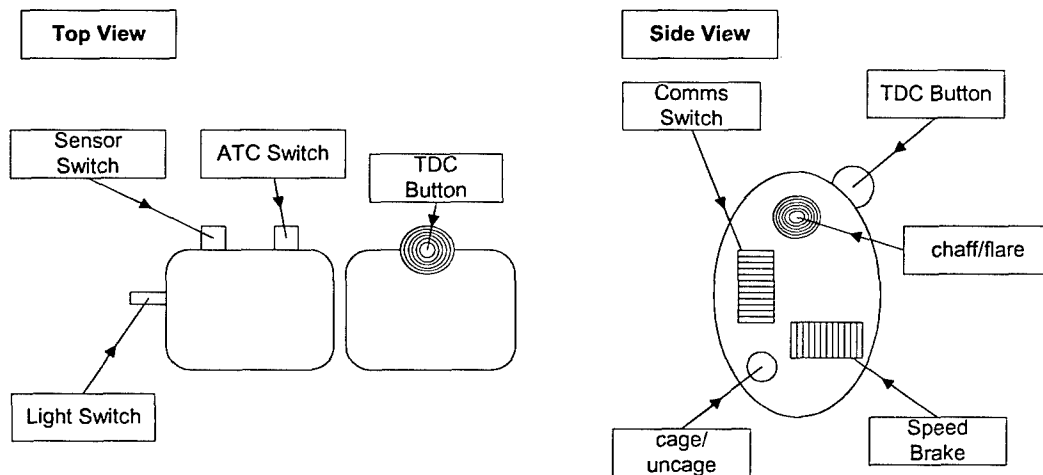


Figure 3. Location of Controls on the Throttle

The Gear Panel (cf18_gearp)

The gear panel differs from the stick and the throttle in that it communicates only with FLSIM. It produces the following discrete values, again defined in cf18_msg_types.h:

Table 12: Discrete Values produced by the Gear Panel

CONTROL	DISCRETE VALUES
Gear Switch	CF18_GEARSWITCH_UP CF18_GEARSWITCH_DN
Flap Switch	CF18_FLAPSWITCH_FULL CF18_FLAPSWITCH_AUTO CF18_FLAPSWITCH_HALF
Parking Brake / Emergency Brake	CF18_PARKBRAKE_ON CF18_PARKBRAKE_EMERG CF18_PARKBRAKE_OFF

This page intentionally left blank

Data Collection

Overview

Data is collected via the router. The data collection program registers with the router, subscribes to message types of interest, and stores messages in text files as comma separated values (CSVs). The program determines a unique session identifier, and this is used to key all information written to the text files, allowing for later retrieval during analysis. The text files are always appended to, never overwritten, reducing the chance of data loss.

Database Design

Although data are stored as CSVs, the architecture itself is designed in accordance with relational database principles. Each text file corresponds to a relation in a database, and the relations are in Boyce-Codd Normal Form. This means that each attribute a relation is either a key of that relation, or functionally dependent upon a key of that relation. An attribute is functionally dependent upon another attribute if its value is determined by that attribute. For example, the author and title of a book are functionally dependent upon the ISBN number, since for each ISBN there is a specific author and title that are determined by it.

Figure 4 shows the structure of the relations. Primary keys are indicated by the '#' sign. All of the relationships are one to many between Session and the other relations.

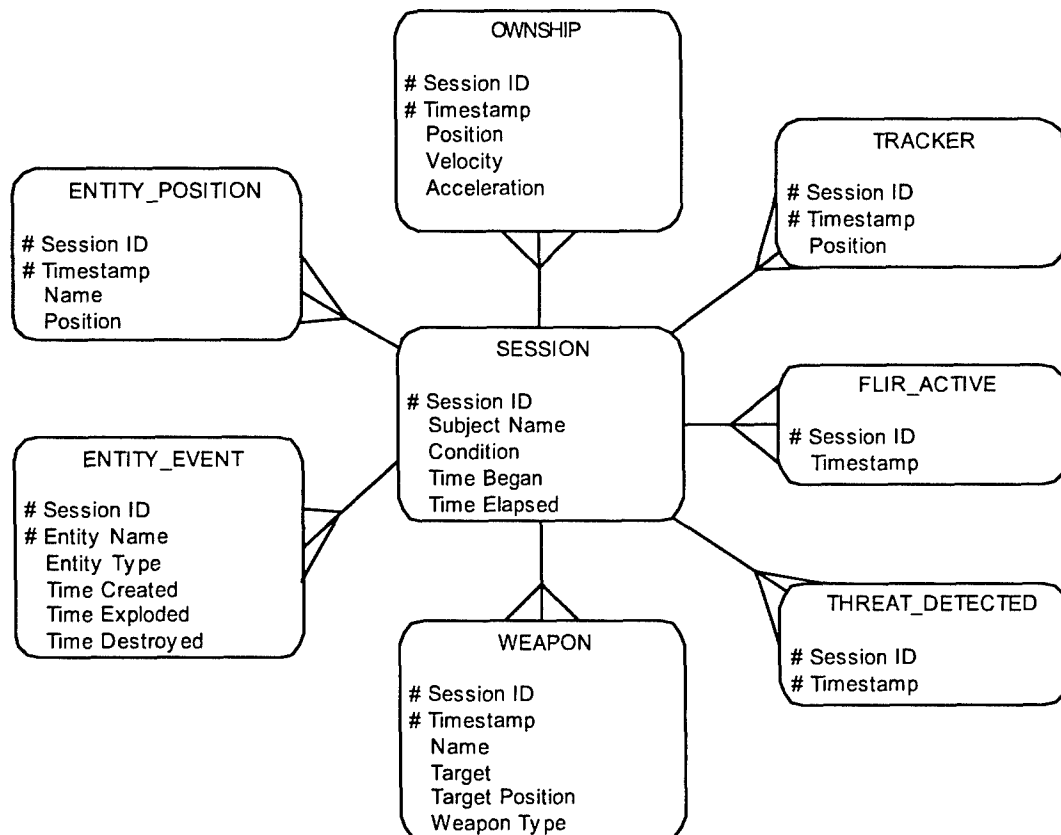


Figure 4. Entity Relationship Diagram

As mentioned above, the data collection program determines a unique session identifier for each experimental trial. This identifier serves a primary key for the session relation, and in foreign key relationships with the other relations. These relationships are illustrated in the following Entity Relationship diagram:

Here is a summary of the relations, giving their purpose, filename, and the message types that they track:

Table 13: Relations

RELATION NAME	PURPOSE	FILENAME	MESSAGE TYPES
SESSION	Store session information.	data/session.txt	None.
ENTITY_EVENT	Store entity creation and destruction events.	data/entity_event.txt	CREATE_ENTITY EXPLODE_ENTITY DESTROY_ENTITY
ENTITY_POSITION	Store entity positional information.	data/entity_pos.txt	POSITION_ENTITY
OWNSHIP	Store positional information for the CF-18.	data/ownership.txt	POSITION_OWNERSHIP
TRACKER	Store head positional information.	data/tracker.txt	TRACKER_UPDATE
FLIR_ACTIVE	Record times that the FLIR is active.	data/flir.txt	FLIR_ACTIVE
THREAT_DETECTED	Record times that a threat was detected.	data/threat.txt	THREAT_DETECTED
WEAPON	Record weapon firing information.	data/weapon.txt	WEAPON_FIRED

Data types are as follows:

- Session ID is an integer.
- Time Began is a character string giving the date and time in the form YYYY-MM-DD HH:MM.
- Timestamp, Time Created, Time Exploded, Time Destroyed and Time Elapsed are all integers giving the number of milliseconds elapsed since the experiment began.
- Position, Velocity, and Acceleration are vectors of the form x, y, z, h, p, r and are floating point values given to a precision of 3 decimal places.
- Target position is a vector of the form x, y, z, which are given to a precision of 3 decimal places.
- The remaining types are character strings.

The Data Collection Program (Datacol)

The data collection program is implemented in C++. It has a multi-threaded design: in general, one thread is allocated to each relation, and has its own file handles and connection to the message router. This allows for messages received to be processed independently of one another. This is done for two reasons: it minimizes the chances of lags in data collection, and it allows a frequency of collection to be assigned to streaming data.

The main function of the data collection program performs following tasks:

1. On start up, the main function reads the session data file and determines the next available session identifier.
2. Once the session identifier is determined, a record containing the session identifier, the subject's name, the experimental condition, and the data and time that the session began is appended to the session data file.
3. A `Timer_c` instance, called *sim_timer* is created to provide a master timer for the simulation. This is used to timestamp all incoming records.
4. Next, the message handling threads are created. The threads are coordinated by means of two mechanisms:
 - A global variable, *done*, is used to control the main loop of each thread. When this is set to true (which occurs when a SIGINT signal is received) each thread exits its main loop and proceeds to clean up its file and router connections.
 - A global *flags* array is maintained. When a thread exits, it sets the corresponding flag in this array to true. The main loop waits on all the flags before exiting to ensure that each thread is given a change to terminate normally.
5. When *done* is set to true, the main loop writes the time the session ended to the session data file, and closes it. It then waits for all of the flags to be set to true before exiting.

Threads handling streaming data maintain their own timer, which is reset every time a message is written to file. Messages are written to file only if a set amount of time has elapsed, which is currently set to be 100 milliseconds. The threads *handle_tracker*, *handle_ownership* and *handle_flir_active* operate in this manner.

The functions generally behave in the following manner:

1. Open output file. If this fails, exit.
2. Connect to router and subscribe to message type of interest. If this fails, exit.
3. Set up the thread timer.
4. Enter run time loop, checking for messages. If less than a set number of milliseconds have elapsed on the thread timer, discard the message. Otherwise, get timestamp from master time, and reset thread timer. Output message to file.
5. On exit, unregister from the router, close output file and set thread completion flag.

The *handle_entity* thread maintains the `ENTITY_EVENT` and `ENTITY_POSITION` relations. It maintains a linked list of *entity_event_s* structures, which track entity creation, explosion and destruction times. It adds to this list only when it receives a `CREATE_ENTITY` message

about an entity type of interest. Currently only targets or threats are considered to be interesting. It identifies these by a naming convention: the name of a target starts with "target", whereas the name of a threat begins with "threat." As it receives POSITION_ENTITY, EXPLODE_ENTITY or DESTROY_ENTITY messages, it acts on them only if the entity name is contained in the list of interesting entities. This prevents a large amount of irrelevant information from being collected. POSITION_ENTITY messages are written to file as soon as they are received. POSITION_ENTITY messages are not currently considered to be streaming, as there are few moving entities in the simulation, and so all messages received are stored. The information tracked by the entity_event_s is written only when data collection ends. This allows for all information about an entity to be stored in one line of the file. Should the program terminate abnormally, this data will be lost.

The remaining thread, *handle_others*, collects data for the WEAPON and THREAT_DETECTED relations. It is functionally similar to the threads handling streaming data, except it does not maintain a timer.

The data collection program is currently has a command line interface, and is invoked as follows:

datacol <subject name> <condition>

Data collection terminates when a SIGINT signal is received, i.e. Ctrl-C is pressed.

This page intentionally left blank

FLSIM

Overview

FLSIM is a Commercial-Off-The-Shelf flight simulation that is used to provide the flight model for the CF-18 ACD. FLSIM is a very extensible product; its functionality can be linked in via a library to a custom application. This allows for the addition of user-defined modules to the simulation. For the ACD, two modifications have been made. One allows for information from the controls to be received via UDP, and the other replaces the FLSIM Database Editor with a user application to control the execution of the simulation.

Both applications are based on samples that came with the FLSIM distribution. Refer to the FLSIM Programmer's Guide for more information.

Simulator Application

The simulator application is called `cf18_flsim_sim`. It takes advantage of FLSIM's extensibility to add code to handle UDP communications from the controls. The interface for the user module to drive the controls is contained in the `flsim_pilot_driver.h` header file. It consists of an interface name, and a large number of pointers to functions that are used to supply the functionality of the user controls. The majority of these functions are quite straightforward, returning specific values from an internal data structure; of interest are `initialize`, `read_device` and `close_device`.

- `initialize` sets up three UDP channels to communicate with the controls, one for the stick, one for the throttle and one for discrete values.
- `read_device` checks for received messages on the UDP channels and update the internal data structure.
- `close_device` closes the UDP channels.

c.f. FLSIM Programmer's Guide, Chapter 5

Control Application

The control application is called `cf18_flsim_con`. It replaces the FLSIM Database Editor as the mechanism for starting and stopping execution for the simulator. This was done, as the full functionality of the Database Editor was not required, and a command line based application was desirable. This also provided a convenient place to export `POSITION_OWNERSHIP` messages, as well as the FLSIM data required to display the DDI's.

An outline of the functionality follows:

1. On commencing execution, the application attempts to connect the FLSIM simulator application. If successful, it initialises a data link, and retrieves control information. Any failures in this part of the application cause the application to terminate.
2. If the FLSIM initialisation section completes successfully, a connection to the message router is attempted. If this succeeds, the application subscribes to HAT (height above terrain) messages.
3. An UDP channel connection is established with the Sims. Flsim_Export_Buffer messages are sent across this, so that the DDI displays can be updated.
4. The fi_runtime_start() function is called to commence the run time loop in the FLSIM simulator application. The control application enters its main loop.
5. In the main loop, a timer is continually checked. If more than 250 milliseconds have elapsed, a POSITION_OWNERSHIP message is sent to the router, and the Flsim_Export_Msg is sent over the UDP connection to the Sims.
6. The main loop is terminated when a SIGINT signal is received (i.e. Ctrl-C is pressed.) or the aircraft crashes. When the main loop exits, it causes the simulation process to end the simulation and exit.

c.f. FLSIM Programmer's Guide, Chapter 7.

Framework

Overview

The simulation framework serves a substitute for STAGE, STRIVE or similar simulation framework development applications. The HMD experiment involved few entities with simple behaviours; so in house development was a viable alternative to the use of a commercial product. There were also requirements for very fine control over the creation of threats. It was much simpler to implement this in custom code, rather than trying to utilize a commercial application.

The MessageQueue_c Class

The MessageQueue_c class is used to enqueue messages to be sent to the message router. The message queue it implements follows a strict First-In First-Out discipline. The method enqueue adds a message to the end of the list, and the method dequeue sends the first message on the list to the router. The constructor for the MessageQueue_c takes a pointer to the IpcClient_c to be used to send messages to the router. The destructor calls the dequeue method repeatedly until no messages are left on the message queue, and then exits. Because of this it is important to delete the MessageQueue_c before deleting the IpcClient_c that is used to send messages to the router.

Table 14: MessageQueue_c Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
MessageQueue_c	pointer to IpcClient_c	Constructor	None.
~MessageQueue_c	None.	Sends all messages in the queue through the router.	None.
enqueue	pointer to IpcMessage_s	Adds a copy of the message to the queue.	None.
dequeue	None.	Removes a message from the queue and sends it to the router.	0 if no message is sent (the queue is empty.) 1 if a message is sent.

The Entity_c Class

An abstract Entity class, called Entity_c is defined in the entity.h header file. All other entity types, and the EntityList_c container type are subclasses of Entity_c. An interface is defined by five methods. The first four methods correspond roughly to the entity update message types: create, update, explode, and destroy. The last method, nameof, provides a means of identifying an entity, which is used during searches.

This architecture was chosen since it allows for new entity types to be added to the simulation relatively easily. As long as the new entity classes are derived from the base Entity_c class, it will not be necessary to change the EntityList_c to support it. The only changes necessary are to the scenario configuration file parser, so that it knows about the new type, assuming that all of the entity's behaviour can be modelled in the update method for it.

In practice, separate EntityList_c objects may be necessary for performance reasons. By separating non-moving entities from moving entities, the calling of update methods for non-moving entities that have no effect can be avoided, for example. Similar changes to the code in interest of efficiency may become necessary.

The StaticTarget_c Class

The StaticTarget_c class is a subclass of the Entity_c class. It is used to model any target that will not move, typically buildings. The methods are defined as follows:

Table 15: StaticTarget_c Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
StaticTarget_c	Entity name, entity type, initial x, y, z, and h position. Pointer to MessageQueue_c to use for enqueueing messages.	Initialises object variables.	None.
~StaticTarget_c	None.	None.	None.
create	None.	Creates and enqueues a CREATE_ENTITY and a POSITION_ENTITY message for this entity.	None.
update	None.	None. (Non-moving entities never update their status.)	None.
explode	None.	Creates and enqueues an EXPLODE_ENTITY message for this entity.	None.
destroy	None.	Creates and enqueues a DESTROY_ENTITY message for this entity.	None.
nameof	None.	Returns the name of the entity.	A constant pointer to the name of the entity.

The StaticThreat_c Class

The static threat class implements the ground and air threats required for the HMD experiment. The experimental requirements stipulated that an air threat would appear at an offset relative to the subject's head position, and remain stationary there for a fixed amount of time.

The methods are defined as follows:

Table 16: StaticThreat_c Methods

METHODS	ARGUMENTS	FUNCTIONALITY	RETURNS
StaticThreat_c	Accepts entity name and type, heading and pitch offsets from head location, threshold distance for appearance, time to live, and pointers to ownship position, tracker position, distance to target and the message queue to use to send messages to the router.	Initialises object variables.	None.
~StaticThreat_c	None.	None.	None.
create	None.	Determines threat location based on subject's head position, and enqueues CREATE_ENTITY and POSITION_ENTITY messages.	None.
update	None.	Checks to see if threshold distance has been passed. If so, calls create method. Now checks against time-to-live. If this has expired, calls destroy method.	None.
explode	None.	None. (Threats cannot be attacked.)	None.
destroy	None.	Enqueues a DESTROY_ENTITY message for this entity.	None.
nameof	None.	Returns the name of the entity.	A constant pointer to the name of the entity.

The EntityList_c Class

The EntityList_c acts as a container for Entity_c and its subclasses. It is implemented as a linked list of pointers to Entity_c objects. The EntityList_c is itself a subclass of Entity_c, and so presents much the same interface. Three additional methods are present, however: add, remove, and find, which define how entities are added, removed and found on the linked list.

The methods are summarized in the following table:

Table 17: EntityList_c Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
EntityList_c	The entity list's name.	Initialises object variables.	None.
~EntityList_c	None.	Removes all nodes from the list, deleting each entity as it proceeds.	None.
create	None.	Calls the create method of every entity on the list.	None.
update	None.	Calls the update method of every entity on the list.	None.
explode	None.	None.	None.
destroy	None.	Calls the destroy method of every entity on the list.	None.
nameof	None.	None.	Constant pointer to the name of the list.
add	A pointer to the entity to add.	Adds an entity to the end of the list.	None.
remove	A pointer to the entity to remove.	Searches through the list, and removes the entity if it is present on the list.	None.
find	The name of the entity to find.	Attempts to find the entity on the list of entities.	A pointer to the entity if successful, otherwise a null pointer.

Scenario Configuration Files

Scenario configuration files obey the following conventions:

- Comments are indicated by '#'
- An entity must be defined on one line.
- Static entities have the following format: <static> <name> <type> <x> <y> <z> <h>. See the constructor for StaticTarget_c. The <static> indicates the entity should be placed on the list of static entities.
- An entity that is to be a mission target has a name that begins with 'target'.
- Threat entities have the following format: <threat> <name> <type> <h> <p> <distance> <ttl>. See the constructor for StaticThreat_c. The <threat> indicates the entity should be placed on the list of threat entities.
- An entity that is a threat has a name that begins with 'threat'.
- The order that the entities are listed in does not matter. However, the last mission target in the list is assumed to be the primary target, and is used when calculating whether a threat should appear.

A sample scenario configuration file is given below:

```
# Scenario File for Condition 1
<static> building_1 house2 20762 -320 0 0
<static> building_2 house1 21291 -315 0 90
<static> target_1 church 20920 -325 0 180
<threat> threat_1 foo -10 15 8500 5
```

The Framework Program

The framework program is itself relatively straightforward. The majority of the simulation is implemented in the various subclasses of Entity_c. The framework program performs the following tasks:

1. Registers with the router, and subscribes to POSITION_OWNESHIP, TRACKER_UPDATE, WEAPON_FIRED and TTI.
2. Creates a new MessageQueue_c to communicate with the router. Creates two EntityList_c classes, one to handle targets, the other to handle threats.
3. Parses the scenario file.
4. Creates two threads to handle messages received from the router, and another to periodically update threats.
5. The main loop consists of repeatedly calling the dequeue method of the MessageQueue_c at 10 millisecond intervals.
6. The check_messages thread operates on the message types as follows:
 - POSITION_OWNESHIP: update the position of the ownship stored in the pos_ownership global variable.
 - TRACKER_UPDATE: update the position of the subject's head stored in the pos_tracker global variable.
 - WEAPON_FIRED: look up the entity fired at based upon name and stores it in the entity global variable. If no entity was the target, 0 is stored in entity instead.
 - TTI: if the value of the TTI message is 0, and the entity exists, the explode method is called for that entity. If the entity doesn't exist, an EXPLODE_ENTITY message is generated with an entity name of 'NULL.'
7. The update_threats thread calls the update method of the EntityList_c handling threats periodically every 250 milliseconds.

Framework Message Types

Information about entities is communicated by four messages types: CREATE_ENTITY, POSITION_ENTITY, EXPLODE_ENTITY and DESTROY_ENTITY.

The data structure for CREATE_ENTITY messages is defined in ipc_msg.h and looks like this:

```
typedef struct {  
    char name[32]; //entity name  
    char type[32]; //entity type  
} CreationMsg_s;
```

There are two naming conventions for entity, as mentioned above: target entities have names beginning with target, and threat entities have names beginning with threat. Entity type is used by the VEGA applications (the FLIR, the HMD) to assign an object to the entity. As such, the entity type must match the name of an object defined in the ADF file used for by the VEGA applications.

POSITION_ENTITY messages have the following data structure:

```
typedef struct {  
    double x, y, z, h, p, r;  
} PosData_s;  
  
typedef struct {  
    char name[32]; //entity name  
    PosData_s loc;  
    PosData_s vel;  
    PosData_s acc;  
} PosUpdateMsg_s;
```

If a module in the simulation receives a message updating the position of an entity for which it has received no CREATE_ENTITY message, the message is discarded. Location, velocity, and acceleration are stored in the three PosData_s structures loc, vel, and acc. Since none of the entities in the HMD experiment move, the velocity and acceleration information is not necessary. However, PosUpdateMsg_s is shared with the POSITION_OWNERSHIP message type, and in the future, it may become necessary to track an entity's velocity and acceleration. The data structure was defined like this, so that it is compatible with STRIVE.

EXPLODE_ENTITY messages contain the following data structure:

```
typedef struct {  
    char name[32];  
} ExplodeEntityMsg_s;
```

Name is the name of the entity that has exploded. If a non-entity is the target, the name is set to NULL, so that a visual explosion effect can still be generated.

Finally, DESTROY_ENTITY messages have this data structure:

```
typedef struct {  
    char name[32];  
} DestroyMsg_s;
```

Name is the name of the entity to remove from the simulation.

Weapons Models

Currently, only an extremely simple laser guided bomb model exists in the Simulation Framework. It is entirely dependent upon the weapon firing mechanisms in the Stores Management Simulation. The bomb goes directly to the target at which it was fired, and explodes when the time-to-impact (TTI) calculated by the Stores Management Simulation reaches zero. The software is flexible enough to allow the insertion of a more sophisticated model.

This page intentionally left blank

Helmet Mounted Display

Overview

The SIMEYE XL-100 Helmet Mounted Display is used to display the Out-The-Window (OTW) scene for the CF-18 ACD. This section briefly discusses the physical characteristics of the SIMEYE, before turning to an in-depth treatment of the VEGA application providing the visual simulation.

SIMEYE XL-100

The SIMEYE provides a field of view of $62^{\circ} \times 48^{\circ}$ at 100% overlap. The use of 100% overlap is recommended in order to reduce eyestrain and simplify rendering. Each projector is capable of displaying 1024 x 768 pixels. The image is provided via a standard RGB cable. The Polhemus tracker was integrated by attaching the tracker transmitter to the underside of the helmet, underneath the foam padding. It was placed directly beneath the centre point of the helmet, but rotated 180° about the x-axis. Due to this, all roll values obtained from the tracker are reversed by 180° . For more details on the SIMEYE, see the owner's manual.

The Scene

A VEGA application is used to generate the out the window scene which is rendered on the helmet mounted display. It uses three channels; one for each eye, and a third to render the FLIR sensor as an overlay over the right eye. The VEGA application runs as three separate processes in order to take advantage of the multiple processors in the Onyx 3200.

Global Data Structures

Two global data structures track the current state of the out-the-window scene. This is necessary, because the VEGA application runs as three separate processes, and therefore needs to set up a shared memory area in order for the processes to utilize a consistent set of information.

The two structures store different sets of information. One, called `otwData`, stores information regarding the state of the CF-18 and the FLIR sensor. It contains information such as the current position, airspeed and altitude of the aircraft, the zoom level of the FLIR, and the range to the current designated target. The other global variable, called simply `GlobalData`, stores pointers to a number of VEGA objects, as well as the router client and a number of runtime timers.

The `otwData` structure is of particular interest, as it contains the values required in order to properly render the symbology set:

```

typedef struct {
    pos_s cf18;      //cf18 position
    pos_s head;      //head position
    pos_s flir;      //flir position

    int airspeed;
    int altitude;

    float aoa;        //angle of attack
    float mach;        //mach number
    float acc;        //acceleration
    float peak_acc;    //peak acceleration

    int ttr_tti;      //0 if TTR, 1 if TTG
    int ttg_seconds;  //number of seconds for TTR/TTG
    int is_designated; //0 if undesignated, 1 if designated

    int wp_number;    //current way point number
    float distance_to_wp; //distance to waypoint
    int master_mode;   //current master mode

    char target_name[32];
    pos_s target_loc;
    float target_bearing;
    float target_elevation;
    float target_range;

    int flir_zoom;
    int flir_mode;

    int blink;        //used to control elements that blink
} otw_data_s;

```

Note that pos_s is defined as a struct containing six floats (x, y, z, h, p, r) containing the x, y, z, heading, pitch and roll of the aircraft. The full definition is in the header file types.h, a copy of which is provided in Annex A.

Head Position and the Polhemus Tracker Client

The OTW scene application connects via shared memory to the Polhemus tracker client in order to determine the current head position. Before the head position is used however, it is adjusted by two offsets. One offset, stored in the GlobalData variable under the values eyeOffset is read from the ADF file, and represents the position of the cockpit. A second group of offsets, defined as constants in the main.cpp file, are used for fine-tuning. These are named DEFAULT_TRACKER_ADJ.

While the application is running, it is possible to reposition the eyepoint using the arrow keys. Pressing the left and right keys toggles through adjusting X, Y, Z, heading, pitch and roll. Pressing up and down increments and decrements the current offset. In order for these changes to be made permanent, it is necessary to adjust either the values defined as constants in the main.cpp file, or to modify the ADF file currently being used.

System Pre-Sync Callback

A system pre-sync callback handles all external information coming to the HMD. It updates the current head position via the shared memory connection to the tracker application, and checks for incoming messages from the message router. The messages it handles are detailed in the following table:

Table 18: HMD Interprocess Communication	
MESSAGE TYPE	ACTION
FLIR_DATA	If the zoom, designate or undesignated buttons were pushed, the corresponding functions are called. Then the x and y values of the TDC button are examined, scaled to a useful magnitude, and noise is eliminated. If they are significant, the otwData global variable is updated, and if the FLIR is in DESIGNATE mode, the designate function is called.
POSITION_OWNERSHIP	The position of the CF-18 is updated in the otwData global variable.
CREATE_ENTITY	The create_entity function is called to add the entity to the linked list containing known entities.
POSITION_ENTITY	The position_entity function is called to update the position of the entity.
EXPLODE_ENTITY	The explode_entity function in the ent_list module is called.
DESTROY_ENTITY	The destroy_entity function in the ent_list module is called.
TTI	Sets ttr_tti in otwData to 1, and sets ttg_seconds to value contained in message.
TT_REL	Sets ttr_tti in otwData to 0, and sets ttg_seconds to value contained in message.

The pre-sync callback also updates the head position in otwData with the current tracker position coordinates read by shared memory from the tracker application. This is offset by the current eye position offset and tracker position offset.

After every iteration, the pre-sync callback recalculates the relative bearing, elevation and range of the designated target, if a target has been designated.

FLIR Simulation

An important component of the Helmet Mounted Display is the FLIR sensor overlay. The simulation is a visual approximation of a FLIR sensor, generated using VEGA SensorWorks and OpenGL. The sensor is implemented using a separate VEGA channel that serves as an overlay over the channel for the right eye of the helmet mounted display. Currently, only the Air-to-Ground mode of the FLIR sensor is emulated.

The FLIR is controlled via buttons on the throttle and stick. The current state of the FLIR is contained in the flir_position, flir_zoom and flir_mode members of the global otwData structure. A summary of the functionality follows:

1. The position of the FLIR is stored in the flir member of the global otwData structure. The x, y and z components are the offset of the FLIR from the aircraft's centre of gravity. The heading and pitch components (or Azimuth and elevation) are controlled via the TDC (Target Designator Computer) button on the throttle. The button functions similarly to a joystick, and reports the amount of movement in a given direction as $\pm x$ and $\pm y$. The TDC button is somewhat noisy, so near zero values are put to zero in the simulation, to avoid a jittering effect.
2. Zoom level is controlled by the ATC Switch on the throttle. Currently, two levels are supported, wide and narrow. Wide is defined as a field of view of 12° by 12° , and narrow is a field of view of 3° by 3° . Field of view is changed in the simulation via the VEGA API call `vgChanFOV`.
3. Mode indicates the level and type of target designation, and controls the sort of FLIR stabilization that is done. When no target is designated, the mode is `SNOW_PLOW` and no stabilization occurs, so that the ground appears to rush by as the aircraft moves. By clicking the TDC button, the pilot designates a point on the ground. The mode becomes `DESIGNATE`, and the FLIR is stabilized to the location on the ground that was designated. This means that as the aircraft moves, the FLIR automatically adjusts its azimuth and elevation so that it remains pointed on the same point on the ground. Designation is released by pushing the undesignate button on the stick. A third mode, `AUTOTRACK`, is left unimplemented. It allows the FLIR to automatically follow a moving target.

A minimal symbology set for the FLIR was developed in OpenGL. The screenshot on the following page illustrates the location of each component.

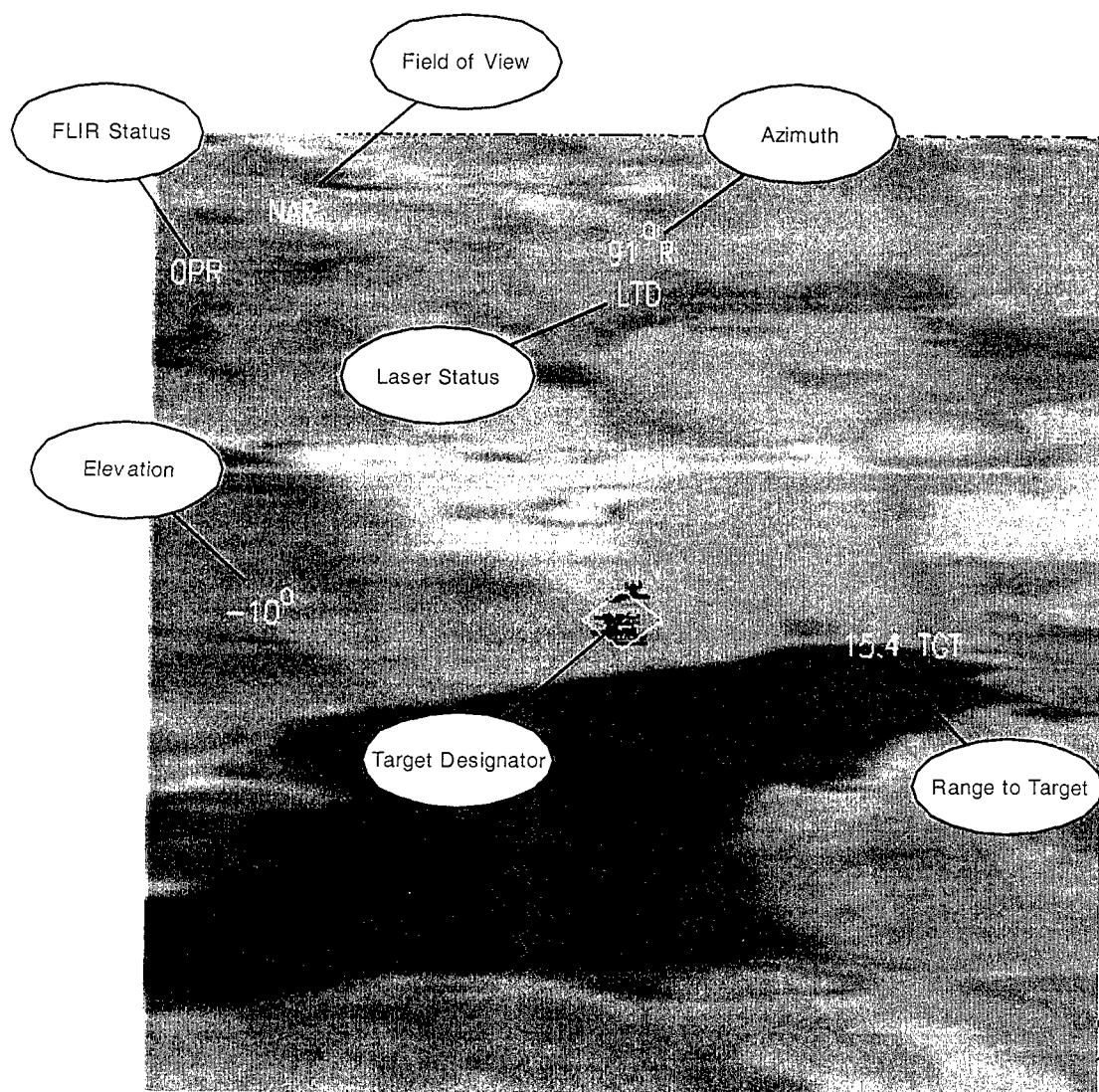


Figure 5. FLIR Symbology Elements

Table 19: FLIR Symbology Elements	
SYMBOLGY ELEMENT	PURPOSE
FLIR Status	Displays 'OPR' if the FLIR is operational, or 'OPR' with two lines through it, if the FLIR is disabled.
Field of View	Displays the current field of view of the FLIR. Wide (12° x 12°) is indicated as 'WIDE', narrow (3° x 3°) is indicated as 'NAR.'
Azimuth	Indicates the current azimuth (heading) of the FLIR. 'R' indicates the azimuth is to the right, 'L' indicates that the azimuth is to the left.
Laser Status	Indicates current laser status. At a distance from the target of greater than 10 nautical miles, or if no target is designated, the laser status is LTD. Within 10 nautical miles, the status changes to L/ARM. When the laser is firing, the status indicator changes to LTD/R, and blinks at a frequency of 5hz.
Range to Target	Indicates the current slant range to the target, in nautical miles. Blank if no target is designated.
Time to Impact	If a weapon has been released, the number of seconds until impact is indicated below the range to the target.
Target Designator	If a target is designated, the target designator appears as a diamond with a dot in the centre (as above.) If no target is designated, the target designator appears as crosshairs. If the FLIR is in the Wide field of view, a rectangle appears around the target designator, indicating what the field of view would be if the FLIR were in Narrow mode.
Elevation	Indicates the current elevation (pitch) of the FLIR sensor.

The implementation of the FLIR symbology is the in the function `draw_flir_symbology`, which is defined in the `flir.cpp` source file. It is registered as a postdraw callback for the FLIR channel. The symbology elements are drawn using OpenGL. The parameters controlling the drawing are stored in the `otwData` global variable, which is passed to the callback through a pointer to void. The functionality of the `draw_flir_symbology` function is summarized below:

1. The `drawSymbology` function commences by saving the rendering state of the VEGA application by calls to `pfPushState`, `pfBasicState`, and `pfPushIdentMatrix`.
2. Next the OpenGL context is set up. The following code fragment sets a view port of 480 x 480 pixels and sets up an orthographic rendering context:

```
glViewport(0, 0, 480, 480);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, 480, 0, 480, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glColor3f(0.0, 1.0, 0.0);
glLoadIdentity();
```

3. After this, the actual drawing calls are made. The line segments are drawn with `GL_LINES`. The VEGA `vgDrawFont` function is used for the text.
4. Once the symbology is drawn, the VEGA rendering context is restored by calls to `pfPopMatrix` and `pfPopState`.

Target Designation

Target designation for ground targets occurs when the TDC button is pressed. It is handled by the designate routine, which performs the following functions:

- It calculates the point of intersection, using a VEGA Isector. The properties of this Isector are set such that it will only intersect with the terrain. If no intersection occurs (i.e. the FLIR is not pointed at the ground) the FLIR state remains SNOW_PLOW.
- If the first Isector intersects with the ground, another Isector is used to try to determine which target has been designated. This Isector is set such that it will only intersect with entities, but not with the terrain. If an intersection is found, the entity name is copied to the target designation message. Otherwise, a null string is used.
- The target position in otwData is updated to the currently designated location, and these values are copied to the target designation message. The message is then sent to the router.

Every time the position of the FLIR changes, the designation routine is called, so that it is possible to move the FLIR when it is in DESIGNATE mode.

The FLIR communicates the designation state by means of two messages: DESIGNATE and UNDESIGNATE. DESIGNATE messages are defined in the DesignateMsg_s data structure, which looks like this:

```
typedef struct {  
    char name[32];  
    double x, y, z;  
} DesignateMsg_s;
```

If the point designated contains an entity then the name contains the entity's name, otherwise the name contains a null string. The x, y, and z values contain the location where the line of sight of the FLIR sensor intersects the ground. UNDESIGNATE messages contain no data.

Height Above Terrain (HAT) Calculation

Height above terrain is calculated by use of an HAT Isector. This information is sent to FLSIM via the message router. At present, this is disabled since the current terrain database's elevation is uniform.

The Entity List

Information about entities is stored in a linked list data structure that is maintained by the ent_list module.

The entity list is based on the following data structure:


```

struct entity_s {
    char name[32];
    char type[32];
    vgPlayer *entity_plyr;
    vgObject *entity_obj;
    entity_s *next;
};

```

The name and type values from the CREATE_ENTITY message are stored in *name* and *type*. The variables vgPlayer and vgObject contain pointers to the VEGA representation of the entity. Next points to the next entity in the list. Ent_list maintains two global pointers to the entity list: one, called first, points to the first node on the list, while current points to the last node on the list, where new entities are added.

Entities are added to the list when CREATE_ENTITY messages are received. The following actions occur:

- A new entity_s is inserted in to the list.
- The name and type of the entity are copied from the CreationMsg_s that was received.
- A new VEGA object is created based on the type of the entity.
- A new VEGA player is created, and the VEGA object is associated with it.

A POSITION_ENTITY message updates the position of the VEGA player associated with the entity.

An EXPLODE_ENTITY message causes current VEGA object associated with the entity to be removed, and to be replaced with a damaged model.

A DESTROY_ENTITY message causes the entity to be removed from the list.

The Router

Overview

The router was designed with the goals of small size, efficiency, and flexibility.

The Router Server

The router was written in Python. Python was chosen because of its fast native support for associative arrays and socket communications. The simplicity of the router lies in the fact that every message the router handles has the same basic structure. It is defined in `ipc_client.h`, and is given below:

```
typedef struct {
    char type[32];
    int size;
} IpcMessageHeader_s;

typedef struct {
    IpcMessageHeader_s hdr;
    char data[BUFFER_SIZE];
} IpcMessage_s;
```

`BUFFER_SIZE` is currently set to 960 bytes. By having a standardized header, the router can handle any message it receives, regardless of the message type.

When the router begins execution, it establishes two UDP receive sockets on ports 9000, and 9001. It listens on port 9000 for control messages, which are of four types: register, unregister, subscribe and unsubscribe, and deal with client management. Port 9001 is used for content messages that will be passed on to subscribers.

It is only necessary for a client to register and subscribe with the router if it intends to receive messages from other clients. Clients producing data, but not receiving it, such as the Polhemus tracker client, can send information directly to port 9001, and it will be passed on to any subscribers to tracker information.

Internally, the tracker maintains two dictionaries, which is the Python term for an associative array, to handle registrations and subscriptions. The *clients* dictionary is indexed by the name of the client, and contains a socket for outward communication to that client. A registration message contains the name, hostname, and port number for a client, which is used to create a new UDP send socket. This socket is then added to the *clients* dictionary, and indexed by the client name. The *messages* dictionary is indexed by message type, and contains a list of clients, which have subscribed to a particular message type. A subscription request is handled by adding the client name to the corresponding list. Unregistration and unsubscribe requests are handled by removing the client from the appropriate dictionaries. The code is

somewhat more complicated than is implied here, because of the need to handle special cases, such as when a message type did not previously exist, or when a client unregisters without unsubscribing from previously subscribed messages.

Content messages that are received are passed on to subscribers by looking up the message type in the messages dictionary, and iterating through the list of clients contained therein. The client is then found in the client dictionary, and the message is sent through the socket associated with that particular client.

Message Router:

Initialize dictionaries for clients and messages.

Initialize list of message types.

Set up sockets for incoming control and message communications.

While not done:

Determine if there are incoming messages.

If incoming control message:

If registration message:

Create socket for new client.

Add socket to clients dictionary.

Send acknowledgement.

If unregistration message:

Send acknowledgement.

Remove client from clients dictionary.

Remove client from messages dictionary.

If subscribe message:

Add client to messages dictionary for that message type.

If unsubscribe message:

Remove client from messages dictionary for that message type.

If incoming message:

Determine message type.

Send message to all clients registered for that message type.

If send connection fails, remove client as described in unregister above.

The Router Client

The client API is essentially a wrapper around the existing UDP communication class, which handles control functions in a more transparent manner than would otherwise be the case. A

IpcClient_c is defined to handle all communication with the router, as well as a generic IpcMessage_s data structure that is used to store all messages to and from the router. The header file ipc_settings.h contains the IP address of the router, the port numbers to connect to, as well as the client's IP address.

The interface to the router is defined through six methods: Register, Unregister, Subscribe, Unsubscribe, Send and Receive, as well as the constructor and destructor. These methods function in a fairly straightforward manner, and are summarized in the following table:

Table 20: Router Client Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
IpcClient_c	A character string for the client name, and an integer for the port.	Creates sockets for sending and receiving messages.	None.
~IpcClient_c	None.	Closes socket connections to router.	None.
Register	None.	Sends registration message to router.	True if router sends acknowledgement.
Unregister	None.	Sends an unregistration to the message router.	True if router sends acknowledgement.
Subscribe	Message type as a character string.	Sends a subscription request to the message router.	True if message sent successfully.
Unsubscribe	Message type as a character string.	Sends an unsubscription to the message router.	True if message sent successfully.
Send	IpcMessage_s structure to send.	Sends a message to the router.	True if message sent successfully.
Receive	IpcMessage_s structure to store received message in.	Checks if a message has been received. (Timeout is set to 2 seconds.)	True if message has been received.

Source code for a Minimal Client Application

The following source code implements an application that communicates with the router. It can serve as a template for future ACD modules.

```
#include <stdio.h>
#include "ipc_client.h"

int main (int argc, char **argv) {

    //display usage if wrong number of command line arguments
    if (argc !=3) {
        printf("Usage: %s <client name> <port>\n", argv[0]);
        return 1;
    }

    int port = atoi (argv[2]);

    IpcClient_c *router = new IpcClient_c(argv[1], port);
```

```

    IpcMessage_s message1, message2;

    if (!router->Register()) {
        printf("Error: Failed to connect to router.\n");
        return 1;
    }

    printf("Client registration was successful\n");

    router->Subscribe("TEST_MESSAGE");

    //set up test message
    sprintf(message1.hdr.type, "TEST_MESSAGE");
    message1.hdr.size = 13;
    sprintf(message1.data, "Hello, World!");

    router->Send(message1);

    //wait for message to be received
    while(1) {
        if (router->Receive(&message2)) {
            printf("Received message:\n");
            printf("%s\n", recvMessage->hdr.type);
            break;
        }
    }

    router->Unregister();
    delete(router);

    return 0;
}

```

The Tracker

Overview

A Polhemus motion tracker device is used to determine head movements relative to a fixed base. The client program communicates with the tracker via a serial port, and sends x, y, z, heading, pitch and roll information by UDP to the message router.

tracker_c

A C++ class, `tracker_c`, handles communication with the Polhemus motion tracker. It sets up a shared memory area, which is accessed by other applications in order to determine the current position of the tracker. The methods are summarized in the following table:

Table 21: *tracker_c* Methods

METHOD	ARGUMENTS	FUNCTIONALITY	RETURNS
<code>tracker_c()</code>	A character string representing the device the tracker is connected to.	Initialises tracker device.	None.
<code>~tracker_c()</code>	None.	Closes tracker device and removes shared memory area.	None.
<code>update()</code>	None.	Polls for a record from the tracker device, and copies it to the shared memory area.	None.
<code>set_verbose()</code>	None.	Toggles verbosity. When verbose, the tracker prints every record read to the standard output.	None.

The `tracker_c` constructor assumes default alignment and hemisphere position, and sets the baud rate to 38400. It sets up the tracker to produce the following record format:

Table 22: Tracker Data Record Format

BYTE	VALUE
0	0 (indicates data record)
1	Tracker station number
2	System Error Code
3 – 6	X
7 – 10	Y
11 – 14	Z
15 – 18	Heading
19 – 22	Pitch
23 – 26	Roll
27 – 28	<cr><lf> (indicates end of record)

The Client Application

The client application is called `tracker`, and is invoked as follows:

tracker <port name>

Optionally, a `-v` option can be specified on the command line, in which case the tracker will operate in verbose mode, printing each record sent to the screen as well as sending it to the router.

The client application creates a `tracker_c` object that handles all communication with the tracker. It maintains a timer, which is used to send tracker positional information to the message router (for data collection purposes.) The frequency that messages are sent is controlled by the `SEND_FREQ` constant.

The Sims

Overview

The Sims provide simulations for the flight systems of the CF-18 ACD. The majority of the simulations are quite straightforward, extracting data from FLSIM and providing it to the VAPS display. Only the Mission Computer, the Stores Management System and the Radar provide much functionality beyond this, and are discussed in detail below. The remaining simulations are summarized in the table below:

Table 23: Sim Classes

NAME	DATA TYPE	FILES	PURPOSE
ADI (Attitude Direction Indicator)	ADI_c	adi.h, adi.cpp	Get ADI data from FLSIM, and copy in to shared memory area.
Audio	Audio_c	audio.h, audio.cpp	Handle communication with an external audio process.
Bezel	Bezel_c	bezel.h, bezel.cpp	Handle bezel presses.
Checklist	CHKLST_c	chklist.h, chklist.cpp	Get checklist data from FLSIM, and copy in to shared memory area.
DDI Manager	DDI_mgr_c	ddi_mgr.h, ddi_mgr.cpp	Manage DDI displays
Engines	ENG_c	eng.h, eng.cpp	Get engines data from FLSIM, and copy in to shared memory area.
FCS (Flight Control System)	FCS_c	fcs.h, fcs.cpp	Get fcs data from FLSIM and copy in to shared memory area.
Mission Computer	MissionComp_c	mc.h, mc.cpp	Process input from controls.
Radar	RDR_c	rdr.h, rdr.cpp	A/A radar simulation
Sim	N/A	sim.cpp, main.cpp	Get data from FLSIM via UDP. Handle execution of other classes in the simulation.
Sim Module	SimModule_c	sim_module.h, sim_module.cpp	Super class for other classes in the Sims. Handles shared memory.

Mission Computer

The mission computer simulation processes messages received from the controls and passes it along to other parts of the ACD. It updates the shared memory pool that is used to communicate with FLSIM. It calls methods of other classes in the Sim to update values (particularly for the Radar and the Stores Management System.) It also sends messages to router.

The mission computer has two modes, based on whether the FLIR is active or not. When the FLIR is active, button presses and streaming TDC data that would normally be passed along to the Stores Management simulation are instead sent to the router in a FLIR_DATA message.

The following table summarizes the action of the mission computer when a particular discrete value is received:

Table 24: Mission Computer Discrete Value Handling

DISCRETE VALUE	ACTION
CF18_WPNSEL_SW_AIM7 CF18_WPNSEL_SW_AIM9 CF18_WPNSEL_SW_GUN	Calls SetAAWeapon in stores management simulation.
CF18_TRIGGER_SW_OFF	Calls FireTriggerRelease method in stores management simulation.
CF18_SENSOR_SW_RGT CF18_SENSOR_SW_LFT CF18_SENSOR_SW_DN	Deactivates FLIR and calls SetTdcHome method in ddi_mgr.
CF18_SENSOR_SW_UP	Activates FLIR.
CF18_WPNREL_SW_OFF	Sets _wpn_rel_dep to false in stores management simulation.
CF18_WPNREL_SW_DEP	Sets _wpn_rel_dep to true in stores management simulation.
CF18_TDCSW_OFF	Calls TdcRelease method in ddi_mgr.
CF18_TDCSW_DEP	If FLIR is active, sets designate to true; otherwise, calls TdcPress method in ddi_mgr.
CF18_COMMSW_OFF	Sets mastermode to Navigation. Calls SetMasterMode method in stores management simulation, and SetRdrMasterMode in radar simulation.
CF18_COMMSW_NO1	Sets mastermode to Air-to-Air. Calls SetMasterMode method in stores management simulation, and SetRdrMasterMode in radar simulation. Sends a MASTER_MODE message to the router.
CF18_COMMSW_NO2	Sets master mode to Air-to-Ground. Calls SetMasterMode method in stores management simulation, and SetRdrMasterMode in radar simulation. Sends a MASTER_MODE message to the router.
CF18_CAGEUNCAGE_OFF CF18_CAGEUNCAGE_DEP	None.
CF18_CHAFFLARSW_OFF CF18_CHAFFLARSW_FLARE	None.
CF18_CHAFFLARSW_CHAFF	Sends a THREAT_DETECTED message to the router.
CF18_ATDSW_DEP	If the FLIR is active, sets zoom to true.
CF18_ATCSW_OFF	None.
CF18_RAIDFLIRSW_DEP	None.
CF18_UNDESIGSW_DEP	If the FLIR is active, sets undesignate to true, and sets _is_designated to false in the stores management simulation; otherwise, calls the BumpAcquisition method in the radar simulation.

Stores Management

In addition to tracking the levels of stores present on the CF-18, the Stores Management Simulation also controls weapons firing.

An environmental variable, CF18_SMS_DATA, contains the location of the Stores configuration file. A CF-18 has nine stations on which weapons can be mounted. On stations 2 and 8 it is possible to mount more than one weapon. The stations on a CF-18 are illustrated in Figure 6, below.

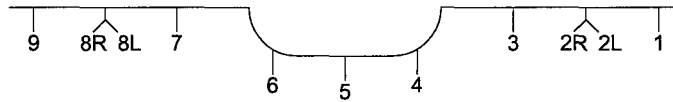


Figure 6. Stations on a CF-18

The Stores configuration file defines the contents of each station. Each line contains information for one of the stations. The format is as follows:

<station> <weapon type> <# of rounds>

Station 0 represents the gun. The available weapon types are GBU12, GBU16, GBU24, AIM7, AIM9, FUEL, and FLIR. FUEL represents a weapon station that is taken up by a fuel container. The FLIR pod also takes up a station, and is normally mounted on station 3.

As mentioned above, the Stores Management system also handles weapons firing. Different firing mechanisms are present for A/A and A/G weapons. The A/A weapons were supported in a previous configuration of the ACD, but are not part of the present experimental requirements. The code implementing them was updated so that it made use of the message router, but the code was otherwise untouched. Previously, there was no support for A/G weapons at all, so this has been added. The remainder of this section discusses A/G weapons only.

The weapon firing mechanism relies upon receiving DESIGNATE and UNDESIGNATE messages from the router. When a DESIGNATE message is received, a copy of the DesignateMsg_s is placed in the member variable _designated_target, and the boolean _is_designated is set to true. An UNDESIGNATE message sets _is_designated to false. The target information is used when generating a WEAPON_FIRED message. The _is_designated must be set in order for a weapon to be fired.

If the ACD is in A/G master mode and a target is designated, the Stores Management simulation immediately begins calculating time-to-go (TTG) and producing TT_REL messages twice per second. These values are determined by the ground range between the CF-18 and the target, and based upon the air speed of the CF-18. TT_REL messages are used to determine when to display the bomb release indicator on the HMD symbology.

When TTG reaches zero, the FireAGWeapon method is called. This method will fire a bomb at the designated target, if the following conditions are true: the master mode is A/G, the CF-18 weapon systems are armed, a target is designated, and the weapon release button has been depressed. If a weapon is fired, a WEAPON_FIRED message is sent to the router, and the `_wpn_fired` flag is set to true. This flag prevents another weapon from being released until the current weapon impacts. Now, instead of TT_REL messages, the Stores Management simulation produces TTI messages containing the time to impact.

TTRelMsg_s and TTIMsg_s define TT_REL and TTI messages respectively. Both contain one variable, seconds, which is a double. WEAPON_FIRED messages have the following structure:

```
typedef struct {
    char name[32];
    char target[32];
    double x, y, z;
    char weapon_type[32];
} WeaponFireMsg_s;
```

Name contains the name of the entity that is firing a weapon. For the CF-18, it is set to ownship. Target is the name of the entity being fired at. X, Y, Z contain the location of the entity being fired at. Weapon type is the type of weapon being released. Currently, it will be one of: LGB (for any bomb), AIM7, AIM9 or GUN.

Radar

The radar simulation receives RADAR_UPDATE messages from the message router and processes the information for the VAPS radar display.

RADAR_UPDATE messages are defined by the following two data structures:

```
typedef struct {
    float srang; //slant range
    float bearing; //relative bearing
    float elevation; //elevation
    float relalt; //relative altitude
    float balt; //barometric altitude
    float tas; //true airspeed
    float cl_rate; //closure rate
    float thdg; //true heading
    float mach; //mach number
    float brg2ownAc; //bearing from tgt to ownship
} RadarTgtInfo_s;

typedef struct {
    char name[32]; //entity name
    char action[32]; //either DELETE or UPDATE
    int tgt_idx; //target index number
    RadarTgtInfo_s tgt_info;
```

} RadarMsg_s;

The radar simulation uses a target index (tgt_idx) internally to track entities. The name is used to produce DESIGNATE messages for the router. The number of targets the radar can track is defined by the constant MAX_TGT. The value is currently 10.

The radar simulation does not determine what entities are visible to it, so any entity passed to it in a RADAR_UPDATE message will be displayed upon the radar. When STAGE was used as the scenario framework, it provided the radar model for the ACD. The Scenario Framework does not currently have a radar model, as it is anticipated that STRIVE will eventually be used to provide this functionality.

This page intentionally left blank

Conclusion

This document has outlined the software architecture of the CF-18 ACD, concentrating on the interprocess communication that ties together the various components of the whole. In particular, the following components were discussed:

- Controls
- Data Collection
- FLSIM
- Scenario Framework
- Helmet Mounted Display
- Message Router
- Polhemus Motion Tracker client
- Simulations.

It is envisioned that as components are added to the CF-18 ACD, this document will need to be revised. Specifically, it is anticipated that the following sections will be added in the next revision:

- Weapons Models
- Scene Generation Interface
- Experimenters Workstation
- VAPS Applications

This page intentionally left blank

Annex A

Header Files

The header files defining the most commonly used data types in the CF-18 ACD are reproduced here for convenience.

ACD_MSG.H

```
#ifndef _ACD_MSG_H
#define _ACD_MSG_H

////////////////////////////////////
// CONSTANTS
////////////////////////////////////
#define MAX_BUF_SIZE 4600

////////////////////////////////////
// STRUCTURES
////////////////////////////////////
typedef struct
{
    int msgType;
    int msgSubType;
    int size;
} AcdMsgHdr_s;

typedef struct
{
    AcdMsgHdr_s    hdr;
    char    data[MAX_BUF_SIZE];
} AcdMsg_s;

#endif
```

CF18_MSG_TYPES.H

```
#ifndef __CF18_MSG_TYPES_H
#define _CF18_MSG_TYPES_H

////////////////////////////////////
// CONSTANTS
////////////////////////////////////
const int MAX_RADAR_TARGETS = 10;

////////////////////////////////////
// ENUMERATIONS
////////////////////////////////////
typedef enum
{
    CF18_MSG_FLT_CTRL = 80,
```



```

    CF18_MSG_SENSOR,
    CF18_MSG_STORES,
    CF18_MSG_EW,
    CF18_MSG_MC,
    CF18_MSG_NAV,
    CF18_MSG_COMM,
    CF18_MSG_DISPLAY,
    CF18_MSG_AUDIO
} CF18_MsgTypes_e;

typedef enum
{
    CF18_FLTCTL_STREAM_STICK_DATA = 8000,
    CF18_FLTCTL_STREAM_THR_DATA,
    CF18_FLTCTL_STREAM_TDC_DATA,
    CF18_FLTCTL_EVENT_DATA
} CF18FltCtltoSimFlsimMsgSubTypes_e;

typedef enum
{
    CF18_TRIMSW_ANU = 8020,
    CF18_TRIMSW_AND,           // 8021
    CF18_TRIMSW_RWD,           // 8022
    CF18_TRIMSW_LWD,           // 8023
    CF18_TRIMSW_OFF,           // 8024
    CF18_SENSORSW_UP,          // 8025
    CF18_SENSORSW_DN,          // 8026
    CF18_SENSORSW_RGT,         // 8027
    CF18_SENSORSW_LFT,         // 8028
    CF18_WPNSELSW_AIM7,        // 8029
    CF18_WPNSELSW_AIM9,        // 8030
    CF18_WPNSELSW_GUN,         // 8031
    CF18_TRIGERSW_OFF,         // 8032
    CF18_TRIGERSW_DET1,        // 8033
    CF18_TRIGERSW_DET2,        // 8034
    CF18_UNDESIGSW_OFF,        // 8035
    CF18_UNDESIGSW_DEP,        // 8036
    CF18_WPNRELSW_OFF,         // 8037
    CF18_WPNRELSW_DEP,         // 8038
    CF18_PADELSW_OFF,          // 8039
    CF18_PADELSW_DEP,          // 8040
    CF18_RAIDFLIRSW_OFF,       // 8041
    CF18_RAIDFLIRSW_DEP,       // 8042
    CF18_SPEEDBRAKE_OFF,       // 8043
    CF18_SPEEDBRAKE_EXT,       // 8044
    CF18_SPEEDBRAKE_RET,       // 8045
    CF18_TDCSW_OFF,            // 8046
    CF18_TDCSW_DEP,            // 8047
    CF18_COMMSW_OFF,           // 8048
    CF18_COMMSW_NO1,           // 8049
    CF18_COMMSW_NO2,           // 8050
    CF18_CAGEUNCAGE_OFF,       // 8051
    CF18_CAGEUNCAGE_DEP,       // 8052
    CF18_CHAFFLARSW_OFF,       // 8053
    CF18_CHAFFLARSW_FLARE,     // 8054
    CF18_CHAFFLARSW_CHAFF,     // 8055
    CF18_ATCSW_OFF,            // 8056
    CF18_ATDSW_DEP,            // 8057
    CF18_IDENTSW_OFF,          // 8058
    CF18_IDENTSW_DEP,          // 8059

```

```

        CF18_RDROPRS_OFF,                // 8060
        CF18_RDROPRS_STBY,              // 8061
        CF18_RDROPRS_OPR,               // 8062
        CF18_RDROPRS_EMERG,            // 8063
        CF18_MASTERARM_ARM,            // 8064
        CF18_MASTERARM_SAFE,           // 8065
        CF18_MASTERMODE_AA,            // 8066
        CF18_MASTERMODE_AG,            // 8067
        CF18_MASTERMODE_NAV,           // 8068
        CF18_LEFTBRAKE_OFF,            // 8069
        CF18_LEFTBRAKE_ON,              // 8070
        CF18_RIGHTBRAKE_OFF,           // 8071
        CF18_RIGHTBRAKE_ON,            // 8072
        CF18_GEARSWITCH_UP,            // 8073
        CF18_GEARSWITCH_DN,            // 8074
        CF18_FLAPSWITCH_FULL,          // 8075
        CF18_FLAPSWITCH_HALF,          // 8076
        CF18_FLAPSWITCH_AUTO,          // 8077
        CF18_PARKBRAKE_OFF,            // 8078
        CF18_PARKBRAKE_ON,             // 8079
        CF18_PARKBRAKE_EMERG           // 8080
    } CF18_FltCtlSwitchTypes_e;

```

```

typedef enum
{
    CF18_RDR_TGT_ADD = 8100,
    CF18_RDR_TGT_DEL,
    CF18_RDR_TGT_UPDT
} CF18_StgToRdrMsgSubTypes_e;

```

```

typedef enum
{
    CF18_RDR_MODE_AA = 8150,
    CF18_RDR_MODE_AG,
    CF18_RDR_MODE_NAV
    // etc...
} CF18_RdrToStgMsgSubTypes_e;

```

```

typedef enum
{
    CF18_WPN_RELEASE = 8200,
    CF18_WPN_JETTISON,
    CF18_WPN_STOP_FIRE
} CF18_SmsToStgMsgSubTypes_e;

```

```

typedef enum
{
    CF18_AUD_GUN_START = 8800,
    CF18_AUD_GUN_STOP,
    CF18_AUD_MISSILE_FIRE_L,
    CF18_AUD_MISSILE_FIRE_R,
    CF18_AUD_CHAFF,
    CF18_AUD_FLARE,
    CF18_AUD_AIM9_SRCH,
    CF18_AUD_AIM9_LOCK,
    CF18_AUD_AIM9_STOP
} CF18_AudioMsgSubTypes_e;

```

```

////////////////////////////////////
// STRUCTURES

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
typedef struct
{
    float srange;
    float bearing;                // relative bearing
    float elevation;
    float relalt;
    float balt;
    float tas;
    float cl_rate;
    float thdg;
    float mach;
    float brg2ownAc;              // relative bearing
} CF18_StgToRdrTargetInfo_s;

typedef struct
{
    int tgt_idx;
    CF18_StgToRdrTargetInfo_s tgt_info;
} CF18_StgToRdrTargetMsg_s;

typedef struct {
    int new_page;
} CF18_MessageToDisplays_s;

typedef struct
{
    int tgt;
    int wpn;
} CF18_SimToStgWpnMsg_s;

typedef struct {
    float stickLat;
    float stickLon;
    float pedal;
    float rightBrakePos;
    float leftBrakePos;
    int trimSw;
} CF18_FltCtl_StickPedalGearP_s;

typedef struct {
    float leftTLA;
    float rightTLA;
    float TDClon;
    float TDClat;
    float antEle;
    float speedBrakePos;
} CF18_FltCtrl_Throttle_s;

typedef struct
{
    float TDClon;
    float TDClat;
} CF18_FltCtrl_TDC_s;

typedef struct
{
    int event;
} CF18_FltCtrl_Discrete_s;

```

```
typedef struct
{
    int event;
} CF18_Audio_s;
```

```
#endif
```

IPC_CLIENT.H

```
#ifndef _IPC_CLIENT_H
#define _IPC_CLIENT_H
```

```
#include "udp_comm.h"
```

```
#include "ipc_settings.h"
```

```
//CONSTANTS
```

```
#define MAX_LEN 1024
```

```
#define BUFFER_SIZE 960
```

```
#define ACK_TIMEOUT 2000
```

```
const int MAX_CONNECT_ATTEMPTS = 5;
```

```
//DATA TYPES
```

```
typedef struct {
    char type[32];
    int size;
} IpcMessageHeader_s;
```

```
typedef struct {
    IpcMessageHeader_s hdr;
    char data[BUFFER_SIZE];
} IpcMessage_s;
```

```
typedef struct {
    char name[32];
    char addr[32];
    int port;
} IpcRegMsg_s;
```

```
typedef struct {
    char name[32];
    char type[32];
} IpcSubMsg_s;
```

```
class IpcClient_c
{
```

```
private:
```

```
    UDPChannel_c* _reg;
    UDPChannel_c* _toRouter;
    UDPChannel_c* _fromRouter;
    int _registered;
    char _name[32];
    int _port;
```

```

    public:
        IpcClient_c(char* name, int port);
        ~IpcClient_c();
        int Register();
        int Unregister();
        int Subscribe(char *type);
        int Unsubscribe(char *type);
        int Send(IpcMessage_s* message);
        int Receive(IpcMessage_s* message);
};

```

```

#endif

```

IPC_SETTINGS.H

```

#ifndef _IPC_SETTINGS_H_
#define _IPC_SETTINGS_H_

#define ROUTER_REG_PORT      9000
#define ROUTER_COM_PORT     9001

const char  DEFAULT_ROUTER_HOST[32] = "acd2";
const char  ROUTER_ADDR[32] = "131.136.72.3";
const char  OWN_ADDR[32] = "131.136.72.3";

#endif

```

LIST.H

```

#ifndef _LIST_H
#define _LIST_H

////////////////////////////////////
// INCLUDES
////////////////////////////////////
#include <stdio.h>

////////////////////////////////////
// STRUCTURES
////////////////////////////////////
struct list_node
{
    struct list_node
        *next,
        *prev;
    int priority;
    void *data;
};

```

```

typedef struct list_node ListNode_s;

/////////////////////////////////////////////////////////////////
// CLASS DEFINITIONS
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
//
// Class:
//
// Purpose:
//
// Created:
//
// Revisions:
//
/////////////////////////////////////////////////////////////////
#ifdef _WIN32_EXPORT
class __declspec(dllexport) List_c
#else
class List_c
#endif
{
    protected:
        ListNode_s *_pHead;
        ListNode_s *_pTail;

    public:

        List_c(void);
        ~List_c(void);
        int _count;

        int IsEmpty(void) { return ( _pHead == NULL); };

        void Empty(void);

        ListNode_s *FirstElement(void);
        ListNode_s *LastElement(void);
        ListNode_s *NextElement(ListNode_s *current);
        ListNode_s *PrevElement(ListNode_s *current);

        void InsertAfter(ListNode_s *after, ListNode_s *pNewNode);
        void InsertBefore(ListNode_s *before, ListNode_s *pNewNode);
        void Enqueue(ListNode_s *pNewNode);
        void Enqueue(void *data, int size);
        void Push(void *data, int size);
        void Push(ListNode_s *pNewNode);

        int Count();

        void Delete(ListNode_s *element);

        void AddNodeWithPriority(ListNode_s *pNewNode, int priority);
        void AddNodeWithPriority(void *data, int size, int priority);
};

/////////////////////////////////////////////////////////////////
// FUNCTION PROTOTYPES
/////////////////////////////////////////////////////////////////

```

```

ListNode_s *CreateNode(void *data, int size);
ListNode_s *MakeNode(void *data, int size);

#endif

```

PERIODIC_PROCESS.H

```

#ifndef _PERIODIC_PROCESS_H
#define _PERIODIC_PROCESS_H

#include <pthread.h>
#include "timer.h"

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

class PeriodicProcess_c
{
protected:

    friend void *PeriodicUpdate(void *arg);

    void (*_pFunc)(void *arg);
    int _period; // msecs
    Timer_c _updateTimer;

    pthread_t _threadID;
    pthread_once_t _doOnce;
    pthread_mutex_t _pauseResume;

    int _done;

public:

    PeriodicProcess_c(void (*func_ptr)(void *), int milliseconds, int
start=1);

    void Start(void);
    void Pause(void) { pthread_mutex_lock(&_pauseResume); };
    void Resume(void) { pthread_mutex_unlock(&_pauseResume); };
    void Kill(void) { _done = TRUE; };
    void Exit(void) { _done = TRUE; };
    void SetPeriod(int milliseconds) { _period = milliseconds; };
    int GetPeriod(void) { return _period; };
    void WaitForCompletion(void) { pthread_join(_threadID, NULL); };
    void DoOnce(void (*func_ptr)(void));
};

#endif

```

SEMAPHOR.H

```

#ifndef SEMAPHOR_H

```

```

#define SEMAPHOR_H

class Semaphore_c
{
    public:

        Semaphore_c(int semid);
        ~Semaphore_c(void) {}

        void Lock(void);
        void Unlock(void);

        void Increment(void);
        void Decrement(void);

        void Destroy(void);

    protected:

        int _SemKey;
        int _SemId;
};

#endif

```

SHM_POOL.H

```

#ifndef SHM_POOL_H
#define SHM_POOL_H

#include "semaphor.h"

#define POOL_NAME_SIZE 32
#define MAGIC_NAME_SIZE 8

#define DEFAULT_DIR_SIZE 100
#define DEFAULT_SHM_SIZE 50000

#define MAGIC_STR "shm_pool"

#define ROUND_TO_64(size,new_size) if ((size) % 8) \
    (new_size) = (size) + (8 - ((size) % 8)); \
    else (new_size) = (size);

enum
{
    NO_LOCK,
    LOCK
};

typedef struct
{
    char name[POOL_NAME_SIZE];
    int offset,
        size;
} DirectoryEntry_s;

```



```

typedef struct
{
    int free_ptr;           // Pointer to the start of free memory
    int number_of_entries; // Size of the directory
    int size;               // Size of the shared memory
    int sem_key;            // Key for the semaphore used to lock this
memory
    char magic[MAGIC_NAME_SIZE];
} DirectoryHeader_s;

class SharedPool_c
{
public:
    SharedPool_c(unsigned int key, unsigned int size,
                  int dir_size=DEFAULT_DIR_SIZE, int sem_key = -1);

    SharedPool_c(unsigned int key);

    ~SharedPool_c(void);

    int Destroy(void);
    int Exists(void);

    void *Allocate(char *name, unsigned int size);
    int Deallocate(char *name);
    void *GetAddressOf(char *name);
    int GetSizeOf(char *name);

    int Size(void) { return _pDirHeader->size; }

    void Lock(void) { _pSemaphore->Lock(); }
    void Unlock(void) { _pSemaphore->Unlock(); }

    DirectoryHeader_s *GetDirectoryHeader(void) { return _pDirHeader; }
    DirectoryEntry_s *Directory(int i) { return &_pDirectory[i]; }

protected:
    int Attach(void);
    int Detach(void);
    int Create(void);
    int Get(void);
    int RemoveExisting(void);

    Semaphore_c *_pSemaphore;

    unsigned int _size;
    unsigned int _key;
    int _shmHandle;

    void *_pLogicalAddress;

    DirectoryHeader_s *_pDirHeader;
    DirectoryEntry_s *_pDirectory;
};

void DumpSegment(DirectoryEntry_s *ent);

#endif

```

TIMER.H

```
#ifndef _TIMER_H
#define _TIMER_H

class Timer_c
{
    int startSeconds;    // Seconds when timer was last reset
    int startUseconds;   // Microseconds remainder when timer was reset

public:

    Timer_c(void) { Reset(); };

    void Reset(void);
    int  ElapsedMilliseconds(void);
    int  ElapsedMicroseconds(void);
    int  ElapsedSeconds(void);
    int  GetCurrentTimeInSeconds(void);
    void WaitMilliseconds(int milli);
};

#endif
```

TYPES.H

```
#ifndef TYPES_H_
#define TYPES_H_

const double DTOR= 0.017453292519943295769236907684;
const double RTOD = 57.295779513082320876798154814105;

enum flir_mode_e {SNOW_PLOW, DESIGNATE, AUTOTRACK};

enum flir_zoom_e {WIDE, NARROW};

enum ttg_e {TTG, TTI};

enum mm_e {A_A, A_G};

typedef struct {
    float x, y, z, h, p, r;
} pos_s;

typedef struct {
    pos_s cf18;    //cf18 position
    pos_s head;    //head position
    pos_s flir;    //flir position

    int  airspeed;
    int  altitude;

    float aoa;      //angle of attack
    float mach;     //mach number
    float acc;      //accelaration
}
```

```

float peak_acc;      //peak acceleration

int ttr_tti; //0 if TTR, 1 if TTG
int ttg_seconds;    //number of seconds for TTR/TTG
int is_designated;  //0 if undesigatged, 1 if designated

int wp_number;      //current way point number
float distance_to_wp;    //distance to waypoint
int master_mode;    //current master mode

char target_name[32];
pos_s target_loc;
float target_bearing;
float target_elevation;
float target_range;

int flir_zoom;
int flir_mode;

int blink;          //used to control elements that blink
} otw_data_s;

#endif

```

UDP_COMM.H

```

#ifndef _UDP_COMM_H
#define _UDP_COMM_H

#include <netinet/in.h> /* sockaddr_in, s_addr */

typedef enum
{
    UDP_ACTIVE,
    UDP_INACTIVE
} UDPState_e;

typedef enum
{
    UDP_RECV_PORT,
    UDP_RECV_ADDR,
    UDP_SEND_PORT,
    UDP_SEND_ADDR,
    UDP_RECV_BUF,
    UDP_SEND_BUF
} UDPAttr_e;

enum
{
    UDP_TRANSCEIVE,
    UDP_SEND,
    UDP_RECEIVE
};

class UDPChannel_c
{
public:

```

```

UDPChannel_c(int mode=UDP_TRANSCEIVE);
~UDPChannel_c(void);

int  RecvMessage(char *msg,int max_len);
int  RecvMessage(char *msg,int max_len,int milliseconds_timeout);

    int  SendMessage(char *msg,int size);

    void Set(int which, int what);
    void Set(int which, char *what);

void Start(void);
void Stop(void);

private:

int  _recvChan;
int  _sendChan;
int  _sendPort,
    _recvPort;
int  _mode;

    int  _recvBufSize;
    int  _sendBufSize;

int  _blockingForIO;

unsigned int  _sendAddr,
            _recvAddr;

char  _ourHostname[128],
      _ourIPStr[64];

UDPState_e  _channelState;

struct in_addr  _ourIP;

struct sockaddr_in  _sendSocketAddr,
                  _recvSocketAddr;

void CreateSendSocket(unsigned int send_addr);
void CreateRecvSocket(unsigned int recv_addr);
};

#endif

```

This page intentionally left blank

Bibliography

Kaiser Electro Optics, Inc., SIM EYE XL 1000 Owner's Manual. (1999)

Virtual Prototypes Inc., FLSIM Programmer's Guide Version 7. (1998)

This page intentionally left blank

List of symbols/abbreviations/acronyms/initialisms

A/A	Air to Air
A/G	Air to Ground
ACD	Aircraft Crewstation Demonstrator
ADI	Attitude Direction Indicator
API	Application Program Interface
COTS	Commercial Off The Shelf
CSV	Comma Separated Value
DRDC Toronto	Defence R&D Canada Toronto
DDI	Digital Display Interface
DND	Department of National Defence
FLIR	Forward Looking Infrared
FOV	Field of View
HAT	Height Above Terrain
HMD	Helmet Mounted Display
HOTAS	Hands on Throttle and Stick
HUD	Heads Up Display
IPC	Interprocess Communication
LGB	Laser Guided Bomb
OTW	Out The Window
RGB	Red Green Blue
TTI	Time To Impact
UDP	User Datagram Protocol

DOCUMENT CONTROL DATA SHEET

1a. PERFORMING AGENCY

DRDC Toronto

2. SECURITY CLASSIFICATION

UNCLASSIFIED

1b. PUBLISHING AGENCY

DRDC Toronto

3. TITLE

(U) Software documentation for CF-18 ACD

4. AUTHORS

Dan Minor, Philip S.E. Farrell

5. DATE OF PUBLICATION

April 1 , 2002

6. NO. OF PAGES

87

7. DESCRIPTIVE NOTES

8. SPONSORING/MONITORING/CONTRACTING/TASKING AGENCY

Sponsoring Agency: DRDC Toronto

Monitoring Agency:

Contracting Agency :

Tasking Agency:

9. ORIGINATORS DOCUMENT NO.

Technical Memorandum TM
2002-026

10. CONTRACT GRANT AND/OR
PROJECT NO.

3ib15

11. OTHER DOCUMENT NOS.

12. DOCUMENT RELEASABILITY

Unlimited distribution but distribution must be pre-approved by DRDC Toronto author

13. DOCUMENT ANNOUNCEMENT

Unlimited announcement

14. ABSTRACT

(U) Since 1998, The Aircraft Crewstation Demonstrator (ACD) has provided the opportunity for scientists and practitioners to review interface designs in a dynamic setting with the human-in-the-loop. This document serves as a reference for the software developed to support the CF-18 ACD that resides at DCIEM.

The ACD has a very modular architecture, which allows for components to be added and removed over time. As such, the ACD is best described in terms of its individual components. The modular nature of the ACD, combined with the physical separation of the software components across several computers, makes interprocess communication of central importance to the software architecture. As such this document gives a comprehensive view of the interprocess communication that occurs during the running of the ACD, before treating each component in depth.

This document is current as of January 17, 2002. It is anticipated that as other components are added to the CF-18 ACD, this reference document will also need to be updated.

15. KEYWORDS, DESCRIPTORS or IDENTIFIERS

(U) software documentation; Aircraft Crewstation Demonstrator; Simulator; Helmet Mounted Display

Defence R&D Canada

is the national authority for providing
Science and Technology (S&T) leadership
in the advancement and maintenance
of Canada's defence capabilities.

R et D pour la défense Canada

est responsable, au niveau national, pour
les sciences et la technologie (S et T)
au service de l'avancement et du maintien des
capacités de défense du Canada.



www.drdc-rddc.dnd.ca