(2)

DTIC
ELECTE
JUN 2 0 1991
S C D

WL-TR-91-3015

URV MULTIPROCESSOR CONTROL SYSTEM AND GROUND STATION CONTROLLER SOFTWARE DESIGN

Capt Michael S. Rottman
Thomas F. Dermis
Lt Amy R. Hartfield
Control Systems Development Branch
Flight Control Division

AD-A237 532

May 1991

Final Report for Period August 1990 – October 1990

Approved for Public Release; Distribution is Unlimited

FLIGHT DYNAMICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6553

91-03598

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

[illegible]
Michael S. Rottman
Project Engineer
Control Systems Development Branch
Flight Control Division

[illegible]
Rudy C. Beavin, Tech Grp Manager
Control Data Group
Control Systems Development Branch
Flight Control Division

FOR THE COMMANDER

[illegible]
H. Max Davis
Assistant For R&T
Flight Control Division

[illegible]
Frank R. Swortzel, Chief
Control Systems Development Branch
Flight Control Division

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/FIGL , WPAFB, OH 45433-6553 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution is Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) WL-TR-91-3015	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Flight Dynamics Directorate; 6b. OFFICE SYMBOL (If applicable) WL/FIGL-1	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Wright Laboratory Wright-Patterson AFB OH 45433-6553	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
62201F	2403	07	50

11. TITLE (Include Security Classification)
URV Multiprocessor Control System and Ground Station Controller Software Design

12. PERSONAL AUTHOR(S)
Capt Michael S. Rottman, Thomas F. Dermis, Lt Amy R. Hartfield

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Aug 90 TO Oct 90	91 May	60

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
09	02	
23	06	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Unmanned Aerial Vehicles, Flight Control Systems, Distributed Control Multiprocessor, Operating Systems, Software Development

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
This report documents the design and implementation of the core controller software for the Multiprocessor Control System (MCS) and Ground Station Core (GSC) for the new Lambda and Gamma Unmanned Research Vehicle (URV) aircraft. Included are the high-level design issues and goals; a description of the real-time multiprocessor operating system (RTMOS) being used; and a complete discussion of the software objects needed for both the MCS and GSC.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Capt Michael S. Rottman	22b. TELEPHONE (Include Area Code) (513)255-8441; 22c. OFFICE SYMBOL WL/FIGL-1

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Foreword

The Flight Control Division of the Air Force Flight Dynamics Directorate, Wright Laboratory has been utilizing unmanned aerial vehicles (UAV's) as flight testbed vehicles since the early 1980's. The focus of the UAV utilization has been to provide a low cost and low risk means to test flight control concepts. One such example of a test performed using the capability was a demonstration of control surface fault detection, isolation and recovery utilizing reconfiguration of control authority gains. The test demonstrated real time performance under actual flight condition. Given the success of the early tests, but also realizing the limitations of the XBQM-106 drones and associated equipment, the Flight Control Division in 1988 began to develop a true testbed capability. This capability, the Unmanned Research Vehicle (URV) Testbed Facility started with the Lambda vehicle, and has begun the evolution into a robust flight testbed.

One of the key design criteria for Lambda was the ability to carry a sufficient embedded electronics system to support not only vehicle control, but also applications under test. With this accomplished, the concepts for the URV Multiprocessor Control System (MCS) were developed. Drawing upon years of experience in multiprocessor systems and software design, engineers at the Flight Control Division envisioned an architecture that would use commercially available components as much as possible, would be transparently expandable, and could be able to collect and telemeter sufficient quantities of data during tests. A goal was to minimize the impacts of hardware change between tests, a process which proved time consuming in previous URV work. To accomplish the above, a key software component, the Real Time Multiprocessor Operating System (RTMOS), was transitioned from another Flight Control Division in-house program, the Advanced Multiprocessor Control Architecture Development (AMCAD) program, where it had been developed and demonstrated. Once the RTMOS had been ported to the MCS architecture, the controller software tasks were designed and developed to control the flow and processing of data through the MCS. The RTMOS and controller software were also adapted for use on the URV Ground Station Core (GSC), an architecture almost identical to the MCS. This report describes the controller software tasks utilized in Lambda aerodynamic data collection flights for both the MCS and GSC.

On 20-21 November 1990, successful flights tests of Lambda, utilizing its new MCS and GSC systems, were conducted. This milestone was achieved after months of dedicated efforts by several engineers. The RTMOS and controller software effort was coordinated by AMCAD project man-

ager, Capt Mike Rottman, and utilized the engineering efforts of Lt Amy Hartfield, Tom Dermis, Jeff Mangen, and Vince Crum. Lt Hartfield, the URV in-house developments manager, also oversaw the integration and test of the MCS and GSC components in preparation for flight test.

Contents

List of Figures

List of Tables

1 Introduction

This report documents the design and implementation of the core controller software for the Lambda and Gamma Unmanned Research Vehicle (URV) Multiprocessor Control System (MCS) and Ground Station Core (GSC). The MCS and GSC architectures will be used for the new Lambda and Gamma aircraft.

The URV electronics architecture, shown in Figure 1, is based on the VMEbus. The MCS will be the on-board control system and data collection unit, and is comprised of processors, signal conditioners, analog-to-digital (A/D) sensor interfaces, pulse width modulated (PWM) servo drivers, and datalink (telemetry) interfaces. Pilot stick command data is transmitted from the GSC to the MCS, where it is routed to the servomotors via the pulse width modulation board. Sensor data from the aircraft surfaces and sensors is converted from analog voltages to 12-bit digital values and downlinked to the GSC.

The GSC architecture is derived from the MCS architecture and uses the same real-time operating system software and many of the same hardware and application software components. The GSC also includes an interface to an Apple Macintosh II workstation, which controls the ground station real-time displays and the storage of flight data on optical disk. The pilot command data is acquired from the pilot's console and sent via telemetry uplink to the MCS. Aircraft sensor data is received from the MCS and stored in a special memory area, called CORE memory, where it is accessed by the Macintosh.

The long-range objective for the MCS and GSC systems is to incorporate control law computation, autopilot functions, and advanced control algorithms into the MCS to provide the envisioned testbed capability. Multiple processors will be used in each system to provide the computational capability needed for advanced, real-time applications. This report documents the initial development and implementation of these systems. For this first phase, the emphasis is on data collection: command data for the aircraft surfaces is uplinked from a human pilot on the ground, and airframe air data and status is downlinked from the MCS to the GSC for later analysis. The controller functions performed on the MCS and GSC systems are responsible for interfacing with each of the hardware devices in the systems to acquire data, perform any necessary conversion and routing of the data, and output the converted data to the appropriate devices. The current MCS and GSC configuration only uses one processor for the core controller functions.

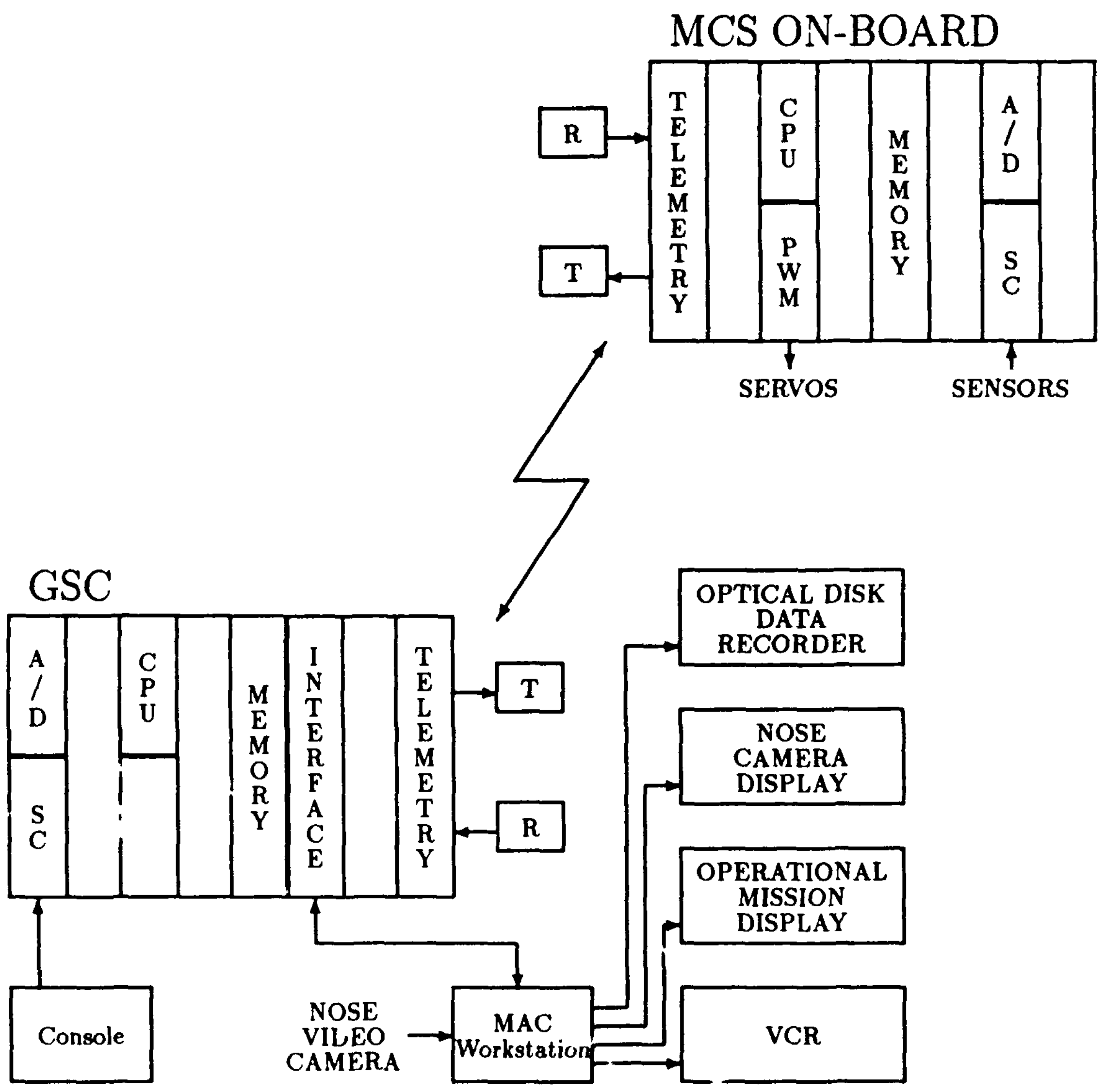

Figure 1: **URV MCS/GSC Architecture**

This report describes the full development of this phase of the MCS and GSC controller software. Included are the high-level design issues and goals, a description of the real-time multiprocessor operating system (RTMOS) being used, and a complete discussion of the software objects needed for both the MCS and GSC. Though this report focuses on the work performed and decisions made for the current implementation, additional detail is provided to serve as a guide for future development and expansion. A detailed description of the telemetry board software and protocols is contained in the Appendix.

2 High-Level Design

2.1 Design Issues

Specific goals of the design are:

1. To the extent possible, use the same software on the MCS and GSC.
2. Each task should perform only one logical function.
3. Tasks should produce their data to specific data variables, without knowledge of which other task or tasks will use this data.
4. Each hardware device should have a dedicated device handler to isolate hardware details from the rest of the software and to reduce the impact of changing devices.
5. Provide a software foundation for the envisioned long term functionality of the MCS and GSC architectures.
6. Make the design flexible and expandable to reduce the overhead of adding processors or changing the application functions to be performed.

For this implementation, the controller software will be limited to the management of the various hardware devices in the systems and the routing of data between them. For example, uplink data received by the MCS must be routed from the telemetry receiver to the pulse width modulation board and to the CORE memory variables. Sensor data from the A/D board must be routed to the CORE memory variables and to the telemetry board to be transmitted to the ground station. No computation, other than data format conversion, will be performed on the data for this phase of development.

Based on the above design goals, unique tasks are defined for each of the hardware interface functions the controller must perform. On the MCS, these interface functions are: PWM, A/D, CORE memory, telemetry input (uplink), and telemetry output (downlink). The GSC functions are A/D, CORE memory, telemetry input (downlink), and telemetry output (uplink). On each system, two additional tasks are defined to perform the actual routing of data between the hardware interface tasks. The Data Collection Input task accepts data from the telemetry receiver and passes it to any tasks which require the information: the CORE and PWM tasks on the MCS and the CORE task on the GSC. The Data Collection Output task performs a similar function for output data; A/D data is routed to the CORE and Telemetry Output tasks on both the MCS and GSC.

This design meets the objectives stated above. Each hardware interface is assigned a unique device handler task which hides the details of the hardware interaction from the rest of the system. This allows changes to the way the device is handled or even a new device without impacting the rest of the software system. Additionally, the rest of the software can be changed without impacting the device handler tasks. All device handler tasks can be used for both the MCS and GSC, as needed. The only task not applicable to the GSC is the PWM task, since there will be no PWM board in the GSC. The overall software task diagrams for the MCS and GSC are shown in Figure 2.

A real-time operating system will be used as the foundation for the controller software. This approach will provide both the real-time performance and the flexibility essential to the MCS and GSC systems. Because the envisioned roles of the MCS and GSC controllers are much greater than needed for this first implementation, the software design must be such that additional functionality can be easily added. Likewise, the design should minimize the impact of adding processors or altering the hardware. A real-time operating system meets these requirements.

2.2 Frame Rate

Real-time software is usually periodic. Various actions must be performed during each period, known as a major frame. The major frame rate for the URV Lambda system is 50 Hz. Within this major frame time, it is desirable to have a number of minor frames to provide sufficient clock resolution for task scheduling and future expansion. Tasks are scheduled on minor frame intervals. Task A, which requires the results of task B, is scheduled for a minor frame following that of task B. The URV controller processor will have 10 minor frames per major frame, for a minor frame rate of 500 Hz. The clock value used by the RTMOS to schedule tasks will therefore be based on the minor frames: each clock tick is equivalent to one minor frame, or 2 milliseconds.

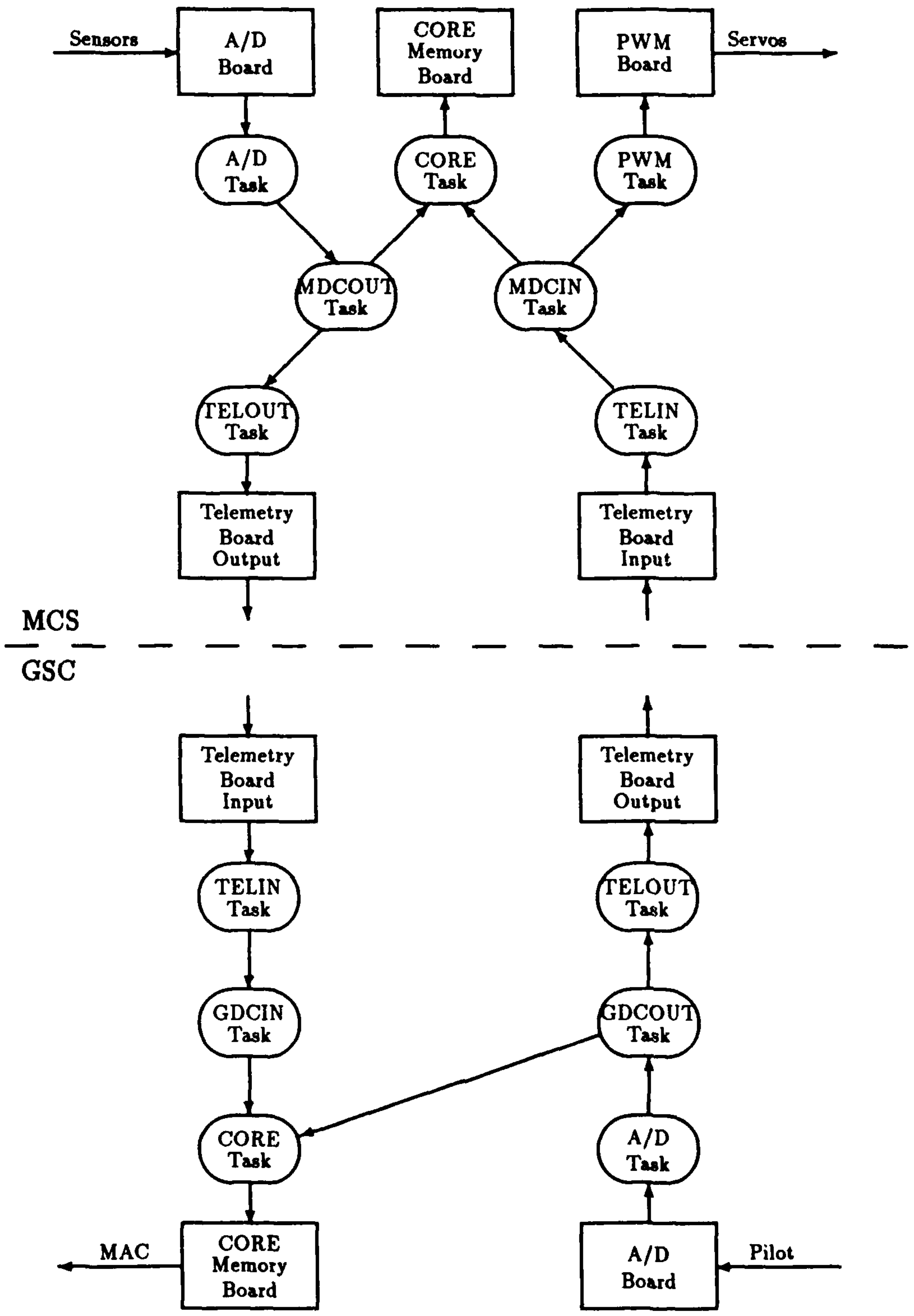

Figure 2: **MCS/GSC Controller Software Tasks and Data Flow**

Based on the tasks needed and the flow of data through the system, different tasks are scheduled at different minor frames within each major frame. The frame task scheduling for the MCS is shown in Figure 3. The tasks to accept uplink data and A/D sensor data are scheduled for the first minor frame. The input and output "data collectors," MDCIN.task and MDCOUT.task, begin in the second minor frame, when the uplink and sensor data should be available. The results of the data collector tasks are sent to the PWM and TELOUT tasks, which are scheduled in the third minor frame. The CORE Update task executes two minor frames later, when all other activity should be completed. The GSC frame scheduling is the same as for the MCS, except there is no PWM task and the data collector tasks are different.

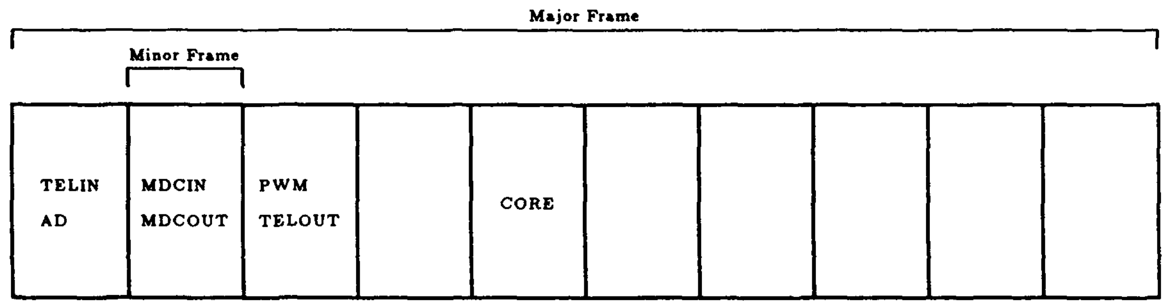

Figure 3: **MCS Major Frame Task Scheduling**

2.3 Initialization

The controller software must initialize the system resources, both on and off the controller CPU board. The RTMOS is responsible for initializing the controller CPU board (clearing RAM, programming the timer) and presetting all operating system data structures. Initialization specific to the application is performed by the controller software. Special code is provided by the application at the beginning of the controller software area for this purpose. This section describes the initializations the applications code will perform.

2.3.1 PWM Board

The pulse width board operates independently of the controller board, reading command data from an on-board data area to send to the control surfaces. Since there is nothing to prevent the PWM board from sampling this data area before the controller task begins producing actual control data from the GSC, the data area must be initialized to some neutral value. In this case, the neutral value (0800h) commands the surfaces to zero degrees. Upon power up, controller software will initialize the data area to that neutral value.

2.3.2 Telemetry

The controller and the telemetry board interact through a set of transmit and receive buffers (two each), with associated flags and semaphores to manage the interaction. Each board is responsible for initializing the flags, semaphores, and buffers that it produces. The controller initialization code will therefore initialize the transmit section. It must set the flags to empty, the semaphores to available, and clear the buffers.

2.3.3 A/D Board

The A/D board interrupts the controller when a sensor channel has completed the conversion of a sampled analog input to digital data. The controller, therefore, must initialize the interrupt controller on the A/D board with the proper interrupt vector and enable the interrupt.

2.3.4 CORE Variables

On cold (power-up) start, the CORE variables are cleared, though this action does not take place on warm processor restart. After a warm processor restart, status information is written to the CORE area which needs to be left intact for later data analysis. The CORE task is responsible for initializing the CORE uplink and downlink type codes the first time it executes (for this implementation).

2.3.5 Global Variables

The global data variables used for intertask communication need to be initialized as well. The producer and consumer flags, the semaphore, and data count variables must all be placed in known states. The data area of each global variable should be initialized to valid starting values for the task. For example, if a task doesn't receive its first data item, it will pass on whatever is already in the global variable, which the task assumes is the previous value of the data. The initialized values must be acceptable "previous data."

2.4 Self-Test

Self-test is performed to determine the health status of the MCS and Ground Station processing elements. For the initial implementation, self-test will occur only on system power up (cold restart), and will be a subset of the test suite envisioned for later implementations. Future versions of the self-test function will include more exhaustive system component testing and a short periodic self-test task to continually reevaluate the system health status in-flight. The full envisioned test suite includes:

1. ROM checksum test.
2. A series of RAM tests.
3. Data movement test.
4. Arithmetic test.
5. Floating point test (if FP is implemented).
6. Logical functions test.
7. Shift and rotate data test.
8. Bit manipulation test.
9. Program control test.
10. Addressing modes test.
11. Test and Set instruction test.
12. Processor registers test.

Initially, only item 1 and item 2 from the above list will be implemented. These two tests do provide some coverage of the other tests just by the nature of their required calculations. A periodic self-test task, as anticipated for the later implementations, must be nondestructive to data, not affect the flying qualities of the aircraft, and complete within a few minor frame times. It is currently envisioned to be the ROM checksum test, as this verifies the integrity of the program code as well as exercising many of the processor logical operations.

2.5 Type Codes

The long-term plans for the URV MCS and GSC include extensive use of data type codes, unique identifications referencing each data type. These type codes greatly increase the flexibility of communications between the two systems, as well as providing a powerful data naming scheme for data collection at the Ground Station Core. In later versions, the RTMOS and controller software will be able to use the data type codes to control data flow, as well as application of specific data items to control law computations, and so on.

The short-term controller design will not support data type codes, however, due to limitations in the bandwidth of the GSC–MCS communications link. The current transmission rate, 125 Kbits per second due to hardware limitations, is just sufficient for uplinking the necessary pilot control data and downlinking the necessary surface and sensor data within the 50 Hz major frame rate. See the Appendix for Telemetry specifics.

To work around the lack of data type codes, the uplink and downlink buffers are of fixed structure, with analog-to-digital data values for specific sensor or pilot command channels in fixed locations in the buffers. For example, the first A/D channel data is stored in the first buffer location, and so on. The data received is loaded into the CORE VARIABLES based on the assumed location in the uplink and downlink buffers. The format for the two buffer types is shown in Figure 4.

3 URV Real-Time Multitasking Operating System

The URV Real-Time Multiprocessor Operating System (RTMOS) is a derivative of the RTMOS developed in FIGL's Advanced Multiprocessor Control Architecture Development (AMCAD) program [5]. In essence, the URV RTMOS is a slimmed down version of the AMCAD RTMOS, with appropriate modifications for the VME-based architecture being used for the URV MCS and GSC. For clarity, the AMCAD RTMOS is simply referred to as the RTMOS, while the URV RTMOS is called URVOS.

Transitioning the RTMOS to the URV was based on two factors. First, the RTMOS has been tested and verified during the past year on the AMCAD testbed. By using proven code, the amount of time needed to debug the URVOS is significantly reduced. Second, the use of an existing in-house RTMOS allows the easy porting of test and application software between the two architectures.

Roll	(a)
Pitch	
Yaw	
Throttle	
Brake	
Switch1	
Switch2	
Switch3	
Switch4	
Switch5	
Switch6	
Switch7	
Switch8	
Error/Status	

Right Flap 1	(b)
Left Flap 1	
Right Flap 2	
Left Flap 2	
Right Aileron	
Left Aileron	
Right Elevator	
Left Elevator	
Right Rudder	
Left Rudder	
Alpha	
Beta	
Alt. Offset	
Lat. Acceleration	
Long. Acceleration	
Vert. Acceleration	
RPM	
Battery Volts	
Alternator Volts	
Roll Attitude	
Pitch Altitude	
Roll Rate	
Pitch Rate	
Yaw Rate	
Airspeed	
Altitude	
Uplink Signal Strength	
Error/Status	

Figure 4: **(a) Uplink and (b) Downlink Buffer Formats**

Emphasis was placed on maintaining the application programming model in porting the RTMOS to the URV for this very reason. It is also historically appropriate that the RTMOS be transitioned to the URV project, since many of the RTMOS concepts were first developed for the URV [6]. These concepts were transitioned to AMCAD, where they were refined and implemented.

In many ways, URVOS is a more straightforward implementation than the RTMOS. The URV's VME-based architecture and global shared memory is much simpler to design for than the AMCAD distributed Virtual Common Memory and assorted protection mechanisms. As a result, the URVOS communications primitives are much shorter and efficient than those used in the RTMOS. Also, the decentralized control algorithms and reconfiguration functions needed by AMCAD are not needed for the URV MCS and GSC, further simplifying the URVOS code.

3.1 Initialization

URVOS initialization is somewhat complicated because of separate system cold start, system warm reset, and warm processor restart conditions. System cold start is the power-up condition. System warm reset occurs when the reset switch on the VME chassis or the reset switch on the VME bus master (the board in slot 0 of the VME chassis) is pressed. When this occurs, all boards in the chassis are reset. The final condition, warm processor restart, occurs when an unexpected interrupt or exception (such as bus error) is encountered.

Each of these conditions may require unique operating system and application initialization sequences, depending on the hardware configuration and specific application. Because of these three conditions, as well as operating system and application initialization, a three phase initialization protocol was developed.

Phase One: Upon power-up (cold system reset), warm system reset, or warm processor restart, the operating system performs self-test. The self-test consists of RAM checks of all RAM used by the URVOS and controller software and a ROM checksum. If self-test is passed, control is passed to the controller cold reset, warm reset, or warm restart software for controller initialization.

Phase Two: Controller reset initializes all off-board hardware devices, such as the PWM and A/D boards, then initializes application data structures. The initialization procedures are described in Section 2.3. Separate initialization routines are used for each of the three reset conditions for maximum flexibility and because the PWM board and CORE variables should only be initialized at cold reset. When the application completes controller initialization, the START system call returns control to the URVOS for phase three initialization.

Phase Three: When all other initialization is complete, the URVOS initializes all hardware devices on the controller processor board and operating system data structures. The warm reset and restart vectors are initialized, scratchpad memory is cleared, the Dual Asynchronous Receiver Transmitter (DUART) is programmed, a free task control block queue is created, and system and application tasks are loaded. Upon completion, the first task is started.

3.2 Taskset Selection

Taskset selection is simple in this implementation because the MCS and GSC systems are currently assumed to have only one processor each performing the control functions. The application software provides a *task tree* which lists the tasks to be loaded into the task management queues. Each application and system task has a header defining the task's starting address, start time, initial priority, and task identification number. The task tree consists of a list of header addresses, terminated with a null (zero) header address. Upon system initialization, the URVOS creates a task control block (TCB) for each task header in the task tree, initializes the TCB with the header information and other initial values, and loads the TCBs into the ready task and timer queues.

Expansion to multiple processors should be very straightforward. The application will provide a task tree for each processor in the system. These task trees will allocate the total task load of the application onto the different processors. Each processor will select a different taskset based on its processor identification number, which will be assigned by the master processor.

3.3 Task Management

A task is a unit of code that executes in parallel with or sequential with respect to other units of code. The task begins executing at the start of its initialization region and ceases execution if it reaches the end of its statement sequence. This means that a task that is intended to run forever must be coded with an explicit loop statement.

In general, the code of a task follows a standard pattern:

1. Initialization,
2. Transaction loop.

The initialization section performs all actions that occur the first time the task executes, to prepare the system for the task's normal execution. For example, it declares and initializes local variables used by the task, performs any handshaking with any other tasks, sends or awaits startup messages, and so on.

The transaction loop then executes repeatedly, as long as the process exists, performing one transaction per iteration. The transaction processing code often resembles the following:

1. Awaits preconditions,
2. Receives input data,
3. Performs computations,
4. Generates output data, and
5. Cleans up and prepares for the next iteration.

The preconditions must be true before the process may proceed with that iteration. Some typical preconditions are: data must be available, a fixed time must have elapsed since the last iteration, a device must change state.

The URVOS provides several system calls which tasks may use to control their execution. The task management system calls are:

1. EXIT
2. SLEEP wakeup_time
3. TERMINATE

The EXIT system call releases the processor, placing the calling task back in the ready queue and causing the highest priority task waiting in the ready queue to be scheduled to execute. If no ready task has a higher priority than the calling task, it will be scheduled again. This system call allows a task which is waiting for some resource or event to release the processor to other waiting tasks. When the task regains the processor, it starts executing after the EXIT call. This is an essential capability, since the URVOS does not support preemptive scheduling. All blocking system calls, such as RECV and P (described below), perform EXITs while they are waiting for the event or timeout.

The SLEEP system call provides the capability for periodic tasking. When a task completes an iteration, it can go to SLEEP until a specified time. Again, the task begins executing after it completes the SLEEP call. In this way, tasks can perform their functions at specific intervals. The

URVOS maintains a global variable, *next_frame*, which tasks can use to sleep until the next major frame or some number of clock ticks into the major frame.

The TERMINATE system call "kills" the calling task and places its TCB back into the free TCB queue.

3.4 Communications

Many real-time operating systems use message passing to implement intertask communications. *Producer* tasks create and send data messages directly to other tasks, which *consume* the data. The flow of data through the application is therefore task-oriented. Task A produces data and sends that data to Task B and Task C, which in turn consume the data and produce new data. This new data is sent to other tasks, and so on. As a result, each task is explicitly linked to the other tasks in the system by means of the data flow. If the application changes, each task may require modification to reflect the new tasks in the system.

The URVOS uses a hybrid message passing/shared memory communications model. Rather than sending messages directly to other tasks, a producer task writes its data to a data structure in shared memory. Tasks needing a piece of data go to the appropriate shared data structure for that data item. This communications model simplifies task design and increases the maintainability of the software. Tasks can be designed without explicit knowledge of or direct interaction with any other task in the system. Tasks are therefore defined by three criteria:

1. what function it performs;
2. what data it produces; and
3. what data it consumes.

A producer/consumer algorithm protects the integrity of the data exchange. When a task produces new data, a producer flag is set. Counting flags can be used to provide frame/data synchronization. Consumer tasks wait until the data is present (has been produced), take the data, then set the consumed flag. This frees the variable to be produced again. This method of intertask communication is a variation of a unilateral rendezvous. Only consumer tasks wait on data, with a timeout set so that wait time is bounded to prevent deadlock. If a producer task attempts to produce new data before previous data has been consumed, the old data will be overwritten but the producer/consumer flags are not updated. If a consumer task times out because the awaited data was never produced, the consumer uses the old data and receives a flag from RECV system call showing that the receive failed.

This involves setting up data structures in shared memory through which tasks pass data or parameters. Every data item passed between tasks is assigned one of these data structures. These data "mailboxes" consist of a semaphore, the data size, a producer flag, a consumer flag, and a count of the number of failed attempts to consume the data. The producer and consumer flags are used to synchronize task interaction to protect data dependencies. The semaphore field, combined with the indivisible "test and set" instruction, provides mutual exclusion on the data structure to prevent tasks from altering the structure concurrently.

Since the URV VMEbus-based architecture has a physical shared memory, true mutual exclusion is possible using semaphore fields. It is possible to allow each data item to have multiple producers and consumers. To remain consistent with RTMOS, however, access to each data structure is limited to one producer task and one consumer task.

Task communication is implemented with two system calls, SEND and RECV, which are used by application tasks to supply data to or acquire data from shared memory. The system calls manage all interaction between the producer and consumer flags in the global data structures, so these interactions are transparent to the application. SEND causes a local data variable to be "sent" to the shared memory, while RECV gets a global variable from the shared memory and puts it in a local variable. The syntax of the commands is:

```
SEND    <SOURCE>,<DESTINATION>,<SIZE>
   where
        SOURCE is the address of the local variable to send
        DESTINATION is the address of the global variable to send to
        SIZE is the number of bytes being sent

RECV    <SOURCE>,<DESTINATION>,<WAIT>
   where
        SOURCE is the address of the global variable to receive
        DESTINATION is the address of the local variable to receive to
        WAIT is the number of clock ticks to wait for the data
```

The source and destination addresses passed to SEND and RECV must be either an address register or an immediate value pointing to the variable. Both SEND and RECV return a flag in the MC68000 data register d7 indicating whether the operation succeeded or failed, based on the producer/consumer algorithm.

In the current, single processor implementation of the MCS and GSC controllers, the shared memory is mapped to the RAM of the controller processor board rather than incorporating a separate shared memory board. When additional processors are added to the controller, the shared memory could be mapped to a separate global memory board simply by changing the address map to correspond to the VME address of the memory board. The shared memory should be a separate card rather than simply mapping it to the local memory of one of the processor cards since the implementation processor cards cannot access their local memory using the VME address. As a result, additional code would be needed to allow that processor to reference the global memory locally, while the other processors would have to do VME references.

3.5 Semaphores

URVOS provides support for semaphores to protect access to resources shared by tasks. Each producer/consumer variable, for example, has a semaphore to prevent the producer and consumer tasks from accessing the data variable concurrently. Semaphore operations can be performed on any defined semaphore in the accessible address space.

Two primitives are provided for semaphore support:

1. P semaphore_address
2. V semaphore_address

The P primitive claims the semaphore; the V primitive releases it.

The semaphore primitives provided by the URVOS are nonblocking. Rather than putting requesting tasks to sleep on a semaphore queue until the resource is available or released (which is difficult to implement in a multiprocessor system), the algorithm uses a form of polling to request the semaphore. The task attempts to claim the semaphore. If it is not available, the task performs an EXIT call to release the processor. In this manner, the waiting task does not prevent other tasks (one of which has already claimed the semaphore) from executing. The P primitive performs these actions, keeping the implementation transparent to the application tasks.

To prevent tasks from busy waiting indefinitely if the semaphore never becomes available, the P primitive has a two clock tick (approximately 4 millisecond) timeout built in. If the task waits for two clock ticks and the semaphore is still not available, the task can proceed without the semaphore. This situation is not ideal, because it could allow improper or incorrect access to shared resources; however, if a task waits forever for a semaphore, deadlock has occurred. When the task claiming the semaphore finishes with the variable or resource, it will reset the semaphore using the V operation. No operating system handling of this error state is currently provided.

4 Multiprocessor Control System Controller Decomposition

4.1 Analog-to-Digital Board Interface Task (AD.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the AD.task.

4.1.1 Overview

This task interfaces with the Analog-to-Digital (A/D) Conversion board. Every major frame the A/D task sends the sensor data collected from the 32 A/D channels by the A/D interrupt handler. The interrupt handler collects the data in the *AD_data_area* on the MCS controller board. The A/D task sends this data to the global *AD_variable*, then triggers the A/D board to begin converting the data from the sensor channels again. When this occurs, the channel number (0) must be written to a memory variable (*Channel*) used by the A/D Interrupt Handler to track which channel its processing.

To abstract the MCS and GCS specific data requirements, the A/D Interrupt Handler and A/D task both assume that all 32 sensor channels are needed. The A/D Interrupt Handler samples all 32 channels, then sends the sensor data "image" to the A/D task, which assumes sensor data is provided for all 32 channels. The determination of how many or which channels are needed is made by the recipient of the data buffer, which consists of 32 data words. An additional error/status word is added after the sensor data, bringing the total buffer length to 33 words.

The algorithm is given in Figure 5.

```
AD.task     tell A/D board to sample channel 0
            write channel number 0 to Channel
            SLEEP until next_frame
            clear error word in AD_buffer
            set RECV timeout to 0
            RECV AD_buffer from ADINT_variable
            if failed
               set error word in AD_buffer
            endif
            SEND AD_buffer to AD_variable
            branch to AD.task
```

Figure 5: **MCS A/D Task Algorithm**

4.1.2 Task Interfaces

This subsection describes the interfaces between the A/D task and the other tasks in the system. The A/D task receives the A/D channel data from the global variable *ADINT_variable*, loading it into a local buffer *AD_buffer*. After checking for error conditions, the buffer data is sent on to the global variable *AD_variable*. The interface is summarized in Table 1.

4.1.3 Error Handling

The A/D task must be able to handle two error conditions. The first and simpler of the two cases is where the sensor data sent by the A/D task is not consumed before new data becomes available. When this occurs, the A/D task overwrites the previous data with the new data message.

Table 1: **MCS A/D Task Interfaces**

Variable Type	Variables
Global variables consumed	*ADINT_variable*
Global variables produced	*AD_variable*
Local variables	*Channel*
	AD_buffer
Off-board variables produced	none
Off-board variables consumed	none

The SEND system call handles this condition automatically. The data never being consumed could indicate that the telemetry link is down — an error case handled by the telemetry board software and the Data Collection Input Task.

The second error condition is potentially more serious. The A/D Interrupt Handler described above should be able to collect a complete sensor buffer in a fraction of the frame rate. The A/D board takes 16 microseconds to convert one channel of sensor data, requiring a total of 512 microseconds to convert all 32 sensor channels (not accounting for interrupt latency). If for some reason the A/D Interrupt Handler has not completed all 32 channels by the end of the frame, the A/D task will pass on the previous frames sensor data, after setting the last item in the buffer (the 33rd) the error flag, to indicate that an A/D error has occurred. Once the data has been sent on, the A/D board is restarted on channel 0.

4.2 Analog-to-Digital Board Interrupt Handler (ADINT.hdlr)

This section presents an overview, the algorithm, the task interfaces, and error handling for the ADINT.hdlr.

4.2.1 Overview

The overall purpose of the Analog-to-Digital Interrupt Handler is to acquire sensor data from the A/D board and build a buffer containing the data for each channel. The A/D board accepts a channel number, reads the sensor data, converts it from an analog voltage to a 12-bit digital data word, then interrupts the controller CPU to indicate that data is available. Conversion of the first channel is triggered each frame by the A/D task, which initializes a memory variable, *Channel*, used by the interrupt handler to "remember" which channel it is processing.

When the controller CPU is interrupted by the A/D board, the interrupt handler reads the channel variable to see which channel the data is from, then writes the data in the *AD_buffer_area*, a fixed block of memory for sensor data collection. The first item in the data area is from A/D channel 0, the second from channel 1, and so on. If all 32 channels have been converted, the interrupt handler performs a SEND system call to send the A/D data to the A/D task. No further conversion is performed until the A/D task triggers conversion of A/D channel 0 at the next frame. Otherwise, the handler triggers the A/D board to begin converting the next channel.

The algorithm is given in Figure 6.

```
ADINT.hdlr    read channel number from *Channel* to register
              if register < 32
                 read channel data from AD board
                 point to *AD_data_area*
                 offset pointer to entry for this channel
                 write channel data to *AD_data_area*
              endif
              increment register            /* point to next channel */
              if register = 32
                 SEND *AD_data_area* to *ADINT_variable*
              else
                 tell AD board to sample next channel
              endif
              save register to *Channel*
              return from interrupt
```

Figure 6: **MCS A/D Interrupt Handler Algorithm**

4.2.2 Interfaces

This subsection describes the interfaces between the A/D Interrupt Handler and the A/D task. The A/D Interrupt Handler loads the channel data into a memory area on the controller board (*AD_data_area*), sending the data to the global variable *ADINT_variable* after all channels have been read. The A/D task will obtain the data from *ADINT_variable*. The A/D Interrupt Handler uses a local variable, *Channel*, to keep track of which A/D channel is currently being convert. The interface is summarized in Table 2.

Table 2: **MCS A/D Interrupt Handler Interfaces**

Variable Type	**Variables**
Global variables consumed	none
Global variables produced	*ADINT_variable*
Local variables	*Channel*
	AD_data_area
Off-board variables produced	none
Off-board variables consumed	none

4.2.3 Error Handling

The interrupt handler has no error cases to handle, assuming that the A/D board will always correctly read the sensor channels and properly perform the analog-to-digital conversion. An error condition handled by the A/D task, described in Section 4.1.3, involves the case where all 32 channels have not been sampled before the next frame. If this occurs, the A/D task resets *Channel* to zero and signals the A/D board to begin conversion of channel zero.

4.3 CORE Variable Update Task (CORE.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the CORE.task.

4.3.1 Overview

The purpose of the CORE Variable Update task is, as the title implies, to update the CORE variables in the shared memory every frame with the latest uplink and downlink data. CORE.task accepts copies of the uplink and downlink data buffers and stores them to the appropriate areas in the CORE variables. For the MCS, the uplink data is a buffer received from the ground station containing pilot stick data. The downlink data is sensor channel data acquired by the MCS A/D board. This task is identical for both the MCS and the GSC.

Though type codes are not being transmitted between the MCS and the Ground Station with the data, the CORE variables still require that type codes be associated with the data items. The controller initialization software can take advantage of the static data buffer formats (Figure 4) and write the appropriate type codes when the task initializes. If a warm processor restart occurs, the task initialization writes the type codes again. In the long term, the CORE Update task will have to be smart enough to handle the type codes with each set of update data.

The algorithm is given in Figure 7.

4.3.2 Task Interfaces

The CORE Variable Update task uses data from two global variables, *uplink_variable* and *downlink_variable*. As expected, the data from *uplink_variable* is used to update the uplink variables in

```
CORE.init    initialize uplink area data type codes
             initialize downlink area data type codes
CORE.task    set RECV timeout to 0
             RECV CORE_buffer from uplink_variable
             if succeeded
                point to beginning of CORE_buffer
                point to beginning of uplink_data_area
                load count of number of items
                call copy_buffer
             endif
             set RECV timeout to 0
             RECV CORE_buffer from downlink_variable
             if succeeded
                point to beginning of CORE_buffer
                point to beginning of downlink_data_area
                load count of number of items
                call copy_buffer
             endif
             SLEEP until next_frame + 4
             branch to CORE.task
```

Figure 7: **MCS CORE Task Algorithm**

the CORE variables. Likewise, *downlink_variable* provides the update information for the downlink CORE variables. The interface is summarized in Table 3.

4.3.3 Error Handling

The general approach in this version of the controller software to handling error cases where an expected data message is never produced is to go ahead with the previous frame's data. With this approach, the tasks further down the line still receive and process data messages, so that only one task sees the error condition. In future implementations, the task detecting the error will flag that condition to some error handling or logging task, but for now the tasks simply proceed with the previous data.

The exception to this approach is tasks that interface directly to other boards, such as the CORE task. Since the CORE variables retain the previous values until overwritten, and can't recognize any difference between data from different frames anyway, there is no point to rewriting the previous values to the CORE values if new ones are not produced.

Table 3: **MCS CORE Update Task Interfaces**

Variable Type	**Variables**
Global variables consumed	*uplink_variable*
	downlink_variable
Global variables produced	none
Local variables	*CORE_buffer*
Off-board variables produced	CORE uplink and downlink areas
Off-board variables consumed	none

4.4 Telemetry Input Handler Task (TELIN.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the TELIN.task.

4.4.1 Overview

The TELIN task reads data from the telemetry receive buffers at the beginning of every major frame. For the MCS, the receive buffers contain uplinked pilot control data from the GSC. TELIN.task uses a subroutine, *get_rx_buffer*, to perform the interaction with the telemetry flags and semaphores. When the data is read, it is sent via *TELIN_variable* to the MCS Data Collection Input task for routing.

The special subroutine (*get_rx_buffer*) is used for two reasons. First, the existing protocol has been coded and verified on the telemetry board. Software was written to test the protocol from the controller perspective, and the subroutine used here will be a conversion of the test routine to a format that can be called from the Telemetry Input task. Second, confining the protocol to a subroutine allows that the protocol to be easily modified without having to change TELIN.task.

The algorithm is given in Figure 8.

4.4.2 Task Interfaces

The interface between the Telemetry Input task and the rest of the system is fairly straightforward. The telemetry board loads uplink data from the GSC into the receiver buffers. TELIN.task attempts to receive uplink data from the telemetry board once every major frame, placing the data in a local workspace, *TELIN_buffer*. After reading the uplink data, TELIN.task sends the data to

```
TELIN.task      point to TELIN_buffer
                call get_rx_buffer
                if previous data available
                   SEND TELIN_buffer to TELIN_variable
                endif
                SLEEP until next_frame
                branch to TELIN.task
```

Figure 8: **MCS Telemetry Input Task Algorithm**

global variable *TELIN_variable* so the routing task (MDCIN.task) can access it. Table 4 shows the interfaces.

4.4.3 Error Handling

When the Telemetry Input task attempts to receive uplink data from the telemetry board, it is possible that when the task checks the buffers, the data has not been produced yet. In this case, since the buffers are empty, nothing is copied from the telemetry board to the task's local workspace. The task therefore still has a copy of data from the previous frame. The telemetry interface subroutine returns a flag signalling the success or failure of the operation. For now, the local buffer is sent whether or not new data was received. Eventually this error condition should be flagged, but for this implementation TELIN.task will simply send on the old data.

Though TELIN.task sends its local buffer whether it contains new or old data, one other case is considered. Before the first uplinked message is received, no previous data is available. To prevent sending "null" buffers, TELIN.task will not begin sending until after the first message has been received. The MCS Telemetry board maintains a counter of the number of messages received that can be checked for this purpose.

Table 4: **MCS Telemetry Input Task Interfaces**

Variable Type	**Variables**
Global variables consumed	none
Global variables produced	*TELIN_variable*
Local variables	*TELIN_buffer*
Off-board variables produced	none
Off-board variables consumed	receive buffers

4.5 Telemetry Output Handler Task (TELOUT.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the TELOUT.task.

4.5.1 Overview

The TELOUT task accepts a message from the MCS Data Collection Output task every major frame, which it loads into the transmit buffers on the telemetry board for downlink to the GSC. TELOUT.task uses a subroutine, *put_tx_buffer*, to perform the interaction with the telemetry flags and semaphores. For the MCS, the transmit data consists of sensor data from the A/D board.

The special subroutine (*put_tx_buffer*) is used for two reasons. For the current implementation, the existing protocol has been coded and verified on the telemetry boards. Software was written to test the protocol from the controller perspective, and the subroutine used here will be a conversion of the test routine to a format that can be called from the Telemetry Input task. Confining the protocol to a subroutine allows the protocol to be easily modified without having to change TELOUT.task.

The algorithm is given in Figure 9.

```
TELOUT.task    set receive timeout to 0
               RECV TELOUT_buffer from TELOUT_variable
               if succeeded
                  point to TELOUT_buffer
                  call put_tx_buffer
               endif
               SLEEP until next_frame + 2
               branch to TELOUT.task
```

Figure 9: **MCS Telemetry Output Task Algorithm**

4.5.2 Task Interfaces

The interface between the Telemetry Output task and the rest of the system is straightforward. TELOUT.task receives downlink data from the MCS Data Collection Output task (MDCOUT.task) through the global variable *TELOUT_variable*, which TELOUT.task places in its local workspace, *TELOUT_buffer*. The task then writes the data to the transmit buffers on the telemetry board. The telemetry board in turn transmits the data to the GSC. Table 5 shows the interfaces.

4.5.3 Error Handling

When the Telemetry Output task attempts to write downlink data to the telemetry board, there is a possibility that when the task checks the buffers, the buffers are not available. In this case, the subroutine will overwrite the buffer with the most recent data. Eventually this error condition should be flagged, but for this implementation the TELOUT task will simply overwrite the old data. The TELOUT task will not write downlink data to the telemetry transmit buffer unless it has received new data.

4.6 MCS Data Collection Input Task (MDCIN.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the MDCIN.task.

4.6.1 Overview

The MDCIN.task acts as an intermediary between the telemetry inputs received from the GSC and any tasks that require the inputs. For now, this task simply serves as a "data exchange," routing data to any task that needs it. No computations are performed, nor are decisions made based on the data received. The uplink data is sent to the PWM task for output to the aircraft servomotors and to the CORE Update task to record the data in the CORE memory uplink variables.

One purpose of this task is to insulate the device handler tasks from the tasks that use the device data. This way, device handlers perform only device-dependent activities: interfacing with

Table 5: **MCS Telemetry Output Task Interfaces**

Variable Type	**Variables**
Global variables consumed	*TELOUT_variable*
Global variables produced	none
Local variables	*TELOUT_buffer*
Off-board variables produced	transmit buffers
Off-board variables consumed	none

some board or device, performing any conversions between device format and a format understood by the rest of the system, and interacting with the system via global mailboxes. This simplifies design of the handlers, and increases their reusability. For example, the PWM.task accepts system data, transforms it to a format expected by the PWM board, and transfers the transformed data to the PWM board. The task does not need to know who produces the data it consumes.

In later versions of the controller, control law computations will need to be performed and additional management functions will be required. The MDCIN.task will have to base its data routing decisions on the data type codes and perhaps perform computations.

The algorithm is given in Figure 10.

4.6.2 Task Interfaces

The MCS Data Collection Input task interfaces with three other tasks in this implementation. All interaction is through the global message variables used for intertask communications. MDCIN.task accepts uplink data from global variable *TELIN_variable*, which was produced by the Telemetry Input (TELIN) task. The data is placed in a local workspace, *MDCIN_buffer*. The uplink data is then sent to *PWM_variable* and *uplink_variable* for use by the PWM task and CORE task, respectively. The interfaces are summarized in Table 6.

4.6.3 Error Handling

The possible error case for MDCIN.task occurs when the expected telemetry data is not produced by TELIN.task. It was decided to simply send on the data from the previous frame, which will still be in the global variable. A later version will record the error case in some manner.

```
MDCIN.task    set receive timeout to 0
              RECV MDCIN_buffer from TELIN_variable
              SEND MDCIN_buffer to uplink_variable
              SEND MDCIN_buffer to PWM_variable
              SLEEP until next_frame + 1
              branch to MDCIN.task
```

Figure 10: **MCS Data Collection Input Task Algorithm**

Table 6: **MCS Data Collection Input Task Interfaces**

Variable Type	**Variables**
Global variables consumed	*TELIN_variable*
Global variables produced	*uplink_variable*
	PWM_variable
Local variables	*MDCIN_buffer*
Off-board variables produced	none
Off-board variables consumed	none

4.7 MCS Data Collection Output Task (MDCOUT.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the MDCOUT.task.

4.7.1 Overview

The MDCOUT.task is an intermediate task between the sensor data from the MCS A/D board and any other tasks in the system that require that data. For now, this task simply serves as a "data exchange," routing the data to any task that needs it. No computations are performed on the data, nor are decisions made based on the type of data received. For this implementation, the A/D sensor data is sent to the Telemetry Output (TELOUT) task for downlink to the GSC and to the CORE Update task so the data can be recorded in the CORE downlink variable area.

One purpose of this task is to insulate the device handler tasks from the tasks that use their data. This way, device handlers perform only device-dependent activities: interfacing with some board or device, performing any conversions between device format and a format understood by the rest of the system, and interacting with the system via global mailboxes. This simplifies design of the handlers, and increases their reusability.

In later versions of the controller, control law computations will need to be performed and additional management functions will be required. The MDCOUT task may also have to route the sensor data to different tasks computing the control law based on the data type codes and other criteria.

The 32 words of sensor data are not all needed by the GSC; only the first 27 channels are used as shown in Figure 4. This task must move the error word (word 33) up to the 28th word, then only send 28 words on to the TELOUT task.

The algorithm is given in Figure 11.

```
MDCOUT.task    set receive timeout to 0
               RECV MDCOUT_buffer from AD_variable
               move error/status word to the 25th word of transmit buffer
               SEND MDCOUT_buffer to downlink_variable
               SEND MDCOUT_buffer to TELOUT_variable
               SLEEP until next_frame + 1
               branch to MDCOUT.task
```

Figure 11: **MCS Data Collection Output Task Algorithm**

4.7.2 Task Interfaces

The MCS Data Collection Output task interfaces with three other tasks in this implementation. All interaction is through the global message variables used for intertask communications. MDCOUT.task accepts sensor data from global variable *AD_variable*, which was produced by the AD.task. The data is placed in a local workspace, *MDCOUT_buffer*. The sensor data is then sent to *TELOUT_variable* and *downlink_variable* for use by the Telemetry Output (TELOUT) task and CORE task, respectively. The interfaces are summarized in Table 7.

Table 7: **MCS Data Collection Output Task Interfaces**

Variable Type	Variables
Global variables consumed	*AD_variable*
Global variables produced	*downlink_variable*
	TELOUT_variable
Local variables	*MDCOUT_buffer*
Off-board variables produced	none
Off-board variables consumed	none

4.7.3 Error Handling

The error case of interest to MDCOUT.task occurs when the expected A/D sensor data is not produced by AD.task. For this implementation, MDCOUT.task simply sends the previous frame's data, which is still in the global variable. A later version will record the error case.

4.8 Pulse Width Modulation Board Interface Task (PWM.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the PWM.task.

4.8.1 Overview

The purpose of this task is to interface with the PWM board. It receives a message containing command data for each of the up to 16 servomotor channels. The data consists of one word per channel, with the first word corresponding to the first channel, and so on. This message is placed in the *PWM_buffer* variable. The PWM task performs any scaling transformations needed to convert the command data (which is in A/D board format) to a format acceptable to the PWM board. When the buffer is ready, data is copied into the PWM board data area. This task executes once every major frame.

For this implementation, the uplink data being sent to the PWM board will use the first 13 PWM channels, as shown in Figure 4. The eight switch values are multiplexed into a single byte value which is sent to the aircraft discretes.

The algorithm is given in Figure 12.

4.8.2 Task Interfaces

This subsection describes the interfaces between the PWM task and the other tasks in the system. The PWM task communicates with the "outside world" via the global data variable *PWM_variable*. The producer of the PWM data (for this version, MDCIN.task) writes the data to *PWM_variable* using the URVOS SEND system call. PWM.task uses the RECV call to copy the data from *PWM_variable* to the local variable *PWM_buffer*. When PWM.task has transformed the data, the buffer is copied to a memory area on the PWM board, *PWM_data_area*. The interfaces are summarized in Table 8.

4.8.3 Error Handling

The only real error condition the PWM task is concerned with for this phase of the URV Controller design involves what to do if the expected buffer is never received. If the expected new

```
PWM.task      set receive timeout to 0
              RECV PWM_buffer from PWM_variable
              if succeeded
                 point to beginning of PWM_buffer
                 point to beginning of PWM_data
                 load count of number of data items
                 call transform_buffer
                 point to beginning of PWM_data
                 point to beginning of PWM_data_area
                 call copy_buffer
              endif
              SLEEP until next_frame + 2
              branch to PWM.task

copy_buffer          while more data
                        get data item from PWM_buffer
                        increment buffer pointer
                        put data item in PWM_data_area
                        increment data area pointer
                     end while
                     return from subroutine
```

Figure 12: **MCS PWM Task Algorithm**

Table 8: **MCS PWM Task Interfaces**

Variable Type	**Variables**
Global variables consumed	*PWM_variable*
Global variables produced	none
Local variables	*PWM_buffer*
Off-board variables produced	*PWM_data_area*
Off-board variables consumed	none

command data is not available, the data from the previous frame will be used to command the servomotors. This case does not require any actions of the PWM task, since the PWM board does not alter the transformed data in its data area. If the data is not available, the task simply goes to sleep until the next frame.

For now this task will only check the data for availability *once*. If the data is not immediately available, the task will go to sleep until next frame. This is accomplished by passing a wait time of 0 to the RECV system call (RECV will only wait for 0 clock ticks for the data to be produced). In the future, the wait time can be changed if desired.

The URVOS provides a easy means of implementing this constraint. The RECV call can be provided with an amount of time to wait for the data to be produced. The call will return a flag indicating whether or not the RECV succeeded (data was available) or failed (none was available).

4.8.4 PWM Data Transformation

The data received by the PWM task (Figure 13 (a)) cannot be passed to the pulse width board immediately. Certain manipulations and computations must be performed to place the data in the appropriate format for use by the servomotors. Figure 13 (b) shows the final form needed for the servomotor commands. The eight switch values are condensed into one byte value which is sent to the PWM discretes. Switch 8 is used to determine whether the four flap values are set to 0800h (neutral flap position, $0°$ deflection) or 0fffh (half flap position, $15°$ deflection).

5 Ground Station Core Controller Decomposition

Most of the controller tasks defined for the Ground Station Core (GSC) are identical to those used in the MCS. This replication is intentional. Similarities between the MCS and GSC requirements were exploited to the greatest extent possible. For tasks that are identical between the two systems, only a brief description of the task is included. Tasks that are different are fully defined.

5.1 Analog-to-Digital Board Interface Task (AD.task)

The GSC Analog-to-Digital Board Interface Task is identical to the one on the MCS, though for the GSC the A/D data processed by this task is pilot command data rather than aircraft sensor

Input Command
Roll
Pitch
Yaw
Throttle
Brake
Switch1
Switch2
Switch3
Switch4
Switch5
Switch6
Switch7
Switch8

(a)

Surface	Command
Left Aileron	Roll
Right Aileron	Roll
Left Elevator	Pitch
Right Elevator	Pitch
Left Rudder	Yaw
Right Rudder	Yaw
Steering	Yaw
Throttle	Throttle
Brake	Pitch
Left Flap 1	0800h/0fffh
Right Flap 1	0800h/0fffh
Left Flap 2	0800h/0fffh
Right Flap 2	0800h/0fffh
Discrete Values	8 Switches

(b)

Figure 13: (a) PWM Task Input Data and (b) Transformed Data Output to PWM Board

data. This task assumes the A/D data is for all 32 channels, even though the pilot commands only use the first 14 channels.

5.2 Analog-to-Digital Board Interrupt Handler (ADINT.hdlr)

The GSC Analog-to-Digital Board Interrupt Handler is identical to the one on the MCS, though for the GSC the A/D data collected by this handler is pilot command data rather than aircraft sensor data. The handler samples all 32 A/D channels.

5.3 CORE Variable Update Task (CORE.task)

The CORE Variable Update task is identical on the GSC and MCS. The task accepts the latest uplink and downlink data every frame and stores them to the appropriate areas in the CORE memory variables. For the GSC, the uplink data is the buffer of pilot stick data from the A/D board. The downlink data is the sensor channel data sent down from the MCS. The GSC CORE variables are accessed by the Macintosh Interface Board to store the data on the optical disk.

5.4 Telemetry Input and Output Tasks (TELIN.task and TELOUT.task)

The two telemetry tasks, TELIN.task and TELOUT.task, which interact with the Telemetry board to pass data between the GSC and the MCS controller CPUs, are identical on both systems. The TELIN task reads data from the telemetry receive buffers at the beginning of every major frame. For the GSC, the receive buffers contain downlinked aircraft sensor data from the MCS. The TELOUT task accepts a message from the GSC Data Collection Output task every major frame, which it loads into the transmit buffers on the telemetry board for uplink to the MCS. For the GSC, the transmit data consists of pilot stick command data from the A/D board.

5.5 GSC Data Collection Input Task (GDCIN.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the GDCIN.task.

5.5.1 Overview

The GDCIN.task is an intermediate task between the telemetry inputs to the Ground Station Core (downlinked from the MCS) and any other tasks in the system that require those inputs. For now, this task simply serves as a "data exchange," routing the data to any task that needs it. No computations are performed on the data, nor are decisions made based on the type of data received. For this implementation, the telemetry downlink data is sent to the CORE Update task so the data can be recorded in the CORE downlink variable area for access by the Macintosh.

In later versions of the controller, many additional functions will be added. The GDCIN.task will have to base its data routing decisions on the data type codes and perhaps perform computations on the data.

The algorithm is given in Figure 14.

5.5.2 Task Interfaces

The GSC Data Collection Input task interfaces with two other tasks in this implementation. All interaction is through the global message variables used for intertask communications. GDCIN.task accepts downlink data from global variable *TELIN_variable*, which was produced by the Telemetry

```
GDCIN.task    set receive timeout to 0
              RECV GDCIN_buffer from TELIN_variable
              SEND GDCIN_buffer to downlink_variable
              SLEEP until next_frame + 1
              branch to GDCIN.task
```

Figure 14: **GSC Data Collection Input Task Algorithm**

Input (TELIN) task. The data is placed in a local workspace, *GDCIN_buffer*. The downlink data is then sent to *downlink_variable* for use by the CORE task. The interfaces are summarized in Table 9.

Table 9: **GSC Data Collection Input Task Interfaces**

Variable Type	**Variables**
Global variables consumed	*TELIN_variable*
Global variables produced	*downlink_variable*
Local variables	*GDCIN_buffer*
Off-board variables produced	none
Off-board variables consumed	none

5.5.3 Error Handling

The possible error case of interest to GDCIN.task occurs when the expected telemetry data is not produced by TELIN.task. It was decided, for this implementation, to simply send on the data from the previous frame, which will still be in the global variable. In a later version, the error case will be recorded in some manner.

5.6 GSC Data Collection Output Task (GDCOUT.task)

This section presents an overview, the algorithm, the task interfaces, and error handling for the GDCOUT.task.

5.6.1 Overview

The GDCOUT.task is an intermediary between the pilot stick data from the A/D board and any tasks in the system that require the stick data. For now, this task simply acts as a "data exchange," routing the data to tasks that need it. No computation is performed on the data, nor are decisions made based on the type of data. For now, the A/D stick data is sent to the Telemetry Output (TELOUT) task for uplink to the MCS and to the CORE Update task so the data can be recorded in the uplink variable area.

In later versions of the controller, control law computations will need to be performed and additional management functions will be required. The GDCOUT.task may have to route the sensor data to additional control law tasks based on the data type codes and other criteria.

Not all 32 words of sensor data are needed by the MCS; only the first 13 channels are used as shown in Figure 4. This task moves the error word (word 33) to the 14th word, then only sends 14 words to the TELOUT task.

The algorithm is given in Figure 15.

```
GDCOUT.task     set receive timeout to 0
                RECV GDCOUT_buffer from AD_variable
                move error/status word to the 14th word of transmit buffer
                SEND GDCOUT_buffer to uplink_variable
                SEND GDCOUT_buffer to TELOUT_variable
                SLEEP until next_frame + 1
                branch to GDCOUT.task
```

Figure 15: **GSC Data Collection Output Task Algorithm**

5.6.2 Task Interfaces

The GSC Data Collection Output task interfaces with three other tasks in this implementation. All interaction is through the global message variables used for intertask communications. GDCOUT.task accepts stick data from global variable *AD_variable*, which was produced by AD.task. The data is placed in a local workspace, *GDCOUT_buffer*. The sensor data is then sent to *TELOUT_variable* and *uplink_variable* for use by the Telemetry Output (TELOUT) task and CORE task, respectively. The interfaces are summarized in Table 10.

Table 10: **GSC Data Collection Output Task Interfaces**

Variable Type	Variables
Global variables consumed	*AD_variable*
Global variables produced	*uplink_variable*
	TELOUT_variable
Local variables	*GDCOUT_buffer*
Off-board variables produced	none
Off-board variables consumed	none

5.6.3 Error Handling

The possible error case for GDCOUT.task occurs when the expected A/D stick data is not produced. It was decided, for this implementation, to simply send on the data from the previous frame, which will still be in the global variable. In a later version, the error case will be recorded in some manner.

6 Decomposition Summary

Table 11 summarizes the tasks that will be used on the GSC and the aircraft MCS. The A/D, Telemetry, and CORE Update Tasks are identical between the two systems. The PWM task is not needed on the GSC since it does not have a pulse width board. The data collection tasks perform similar functions on their respective boards, but are slightly different internally since they send copies of the A/D data to different CORE variables.

Table 11: **MCS and GSC Task Summary**

MCS Tasks	GSC Tasks
AD.task	AD.task
CORE.task	CORE.task
TELIN.task	TELIN.task
TELOUT.task	TELOUT.task
MDCIN.task	GDCIN.task
MDCOUT.task	GDCOUT.task
PWM.task	

References

[1] Mizar/Integrated Solutions. *MZ 8115 68000/68010-Based CPU Module User's Manual, Eighth Edition.* Technical Report 7101-00025-0001. Mizar/Integrated Solutions, Carrollton TX, September 1989.

[2] Mizar/Integrated Solutions. *MZ 8605 Analog Input Module User's Manual.* Technical Report 7101-00033-0001. Mizar/Integrated Solutions, Carrollton TX, September 1989.

[3] Motorola. *M68000 Microprocessor User's Manual, Seventh Edition.* Englewood Cliffs NJ: Prentice-Hall, 1989.

[4] Motorola Semiconductor Products, Inc. *MC68681 Dual Asynchronous Receiver/Transmitter (DUART).* Technical Report ADI-988. Motorola Semiconductor Products, Inc., Austin TX, October 1983.

[5] Rottman, Michael S. and Daniel B. Thompson. "The AMCAD Real-Time Multiprocessor Operating System." In *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference*, pages 1813–1818, New York: IEEE Press, 1989.

[6] Thompson, Daniel B. *A Multiprocessor Avionics System for an Unmanned Research Vehicle.* Technical Report AFWAL-TR-88-3003, Wright-Patterson AFB OH: AFWAL/FIGLB, March 1988.

A Telemetry

A.1 Overview

The transfer of accurate data messages is extremely important when flying the Unmanned Research Vehicle (URV). The messages sent to the Multiprocessor Control System (MCS) on the aircraft contain the command words used to control the URV servomotors, and the messages sent to the Ground Station (GSC) contain the A/D values collected from the onboard sensors. A telemetry system has been designed to ensure these messages are transmitted and received correctly.

The telemetry system operates autonomously, executing its functions on dedicated processing boards. Flying the URV requires two telemetry boards: one on the ground, controlling transmissions to and from the GSC, and another in the URV, controlling transmissions to and from the MCS (Figure A–2). Each board is a MC68000 CPU module, responsible for handling both incoming and outgoing messages. Outgoing messages originate at the main processing unit and are transferred to the telemetry board using the VME backplane. The telemetry board sends these messages serially to the microwave transmitter using its onboard Dual Asynchronous Receiver Transmitter (DUART). Similarly, an incoming message is transferred serially from the microwave receiver to the telemetry board, and then read by the main processing unit via the VME backplane.

The format for telemetering data is a handshaking protocol in which the receiving board always answers the transmitting board. When there are no errors, the overall transmitting sequence should look like Figure A–3. First, the GSC sends a block of data, then the MCS responds by returning status. Next, the GSC commands the MCS to downlink, then the MCS transmits a block of data. Finally, the GSC returns status. This sequence should continually loop, with the GSC always initiating the process. In this setup, the GSC acts as the bus master, because it directs the priority of data transmissions. It controls its own transmissions and also determines when the MCS is allowed to downlink.

The telemetry boards transmit and receive at 125 Kbaud. However, since the MCS and GSC do not transmit and receive simultaneously, the total bandwidth is 208 bytes per major frame. In the current configuration, the MCS needs to transmit 61 bytes per major frame and the GSC needs to transmit 32 bytes. These needs only utilize about half of the total bandwidth, and make it possible to initiate a single retry each major frame. The retry forces a second transmission of the

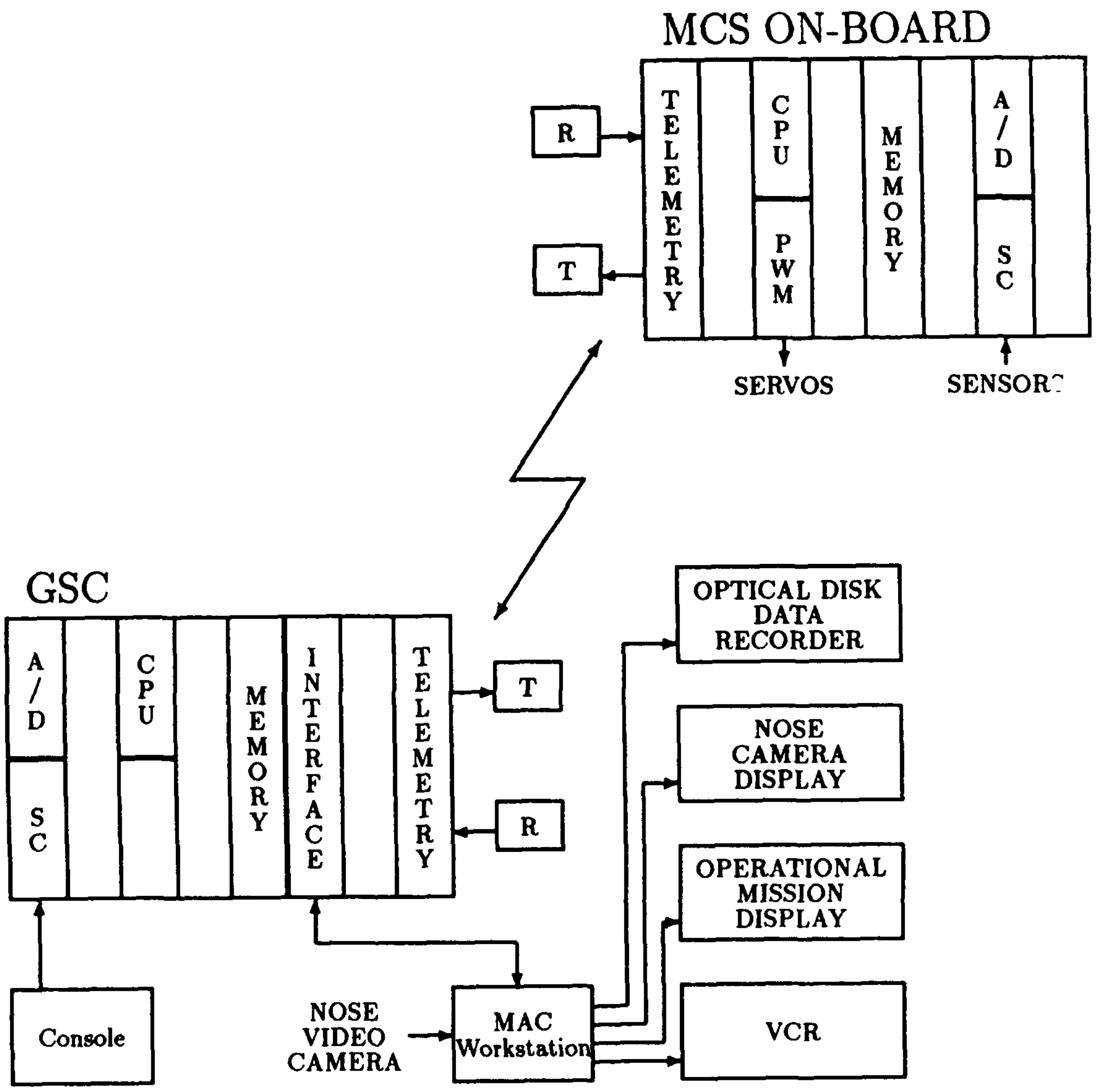

Figure A-2: **URV MCS/GSC Architecture**

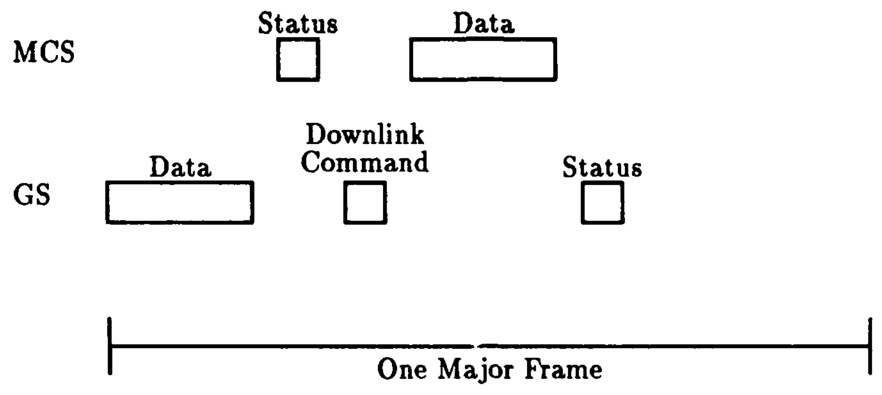

Figure A–3: **Telemetry Handshaking**

current data block. If the returned status byte indicates that the previous data transmission was erroneous, then the sender of the data will transmit it again. After two incorrect transmissions of the same data block, the block is "scrapped," and overwritten with new data.

The software implementing the protocol contains two main sections: a receive section and a transmit section. The receive section is implemented using an interrupt handler, and the transmit section is actually the main program. Aside from the few additions to make the GSC the bus master, the MCS and GSC implement the protocol almost identically.

A.2 Receiving

The interrupt handler controls all incoming messages. The messages are transferred from the telemetry board's DUART one byte at a time. When the DUART receives a new byte, it generates an interrupt. The interrupt handler then stores and interprets this byte. It will first determine if the byte is the beginning of a new message, or a continuation of a previous message. There are three different types of messages:

1. a command-to-downlink-data message,
2. a data message, or
3. a status message.

The first and third message types are always one byte each, but the second type can be of variable length, depending on the number of data bytes. Figure A–4 shows the message types.

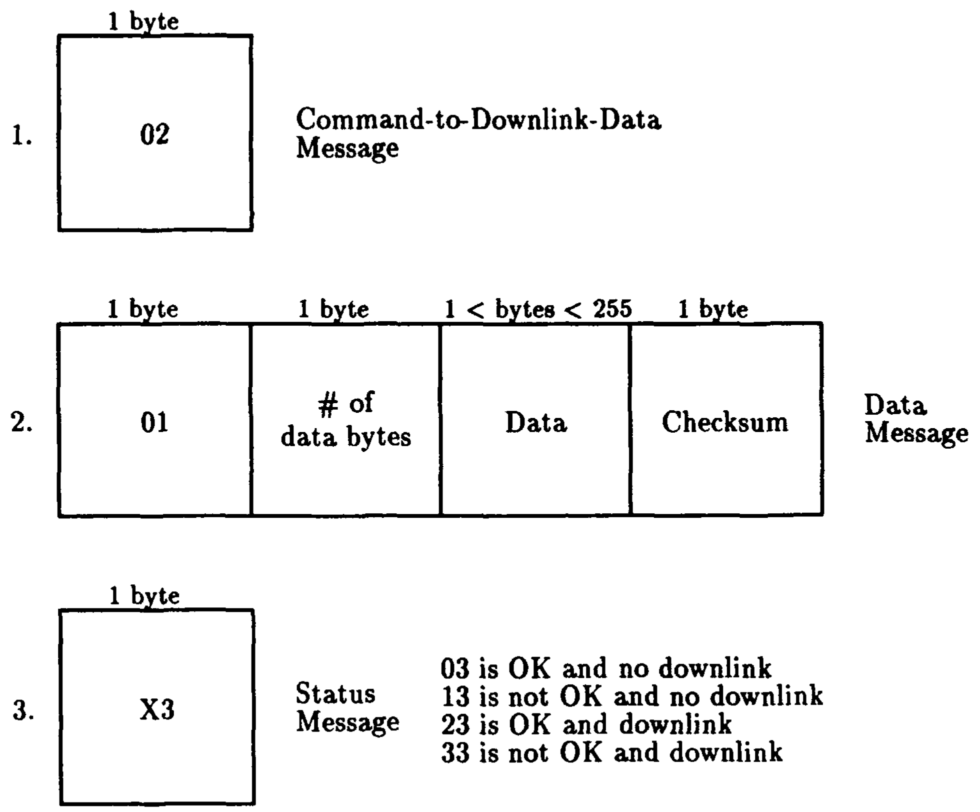

Figure A-4: **Message Types**

The first type of message, the Command-to-Downlink-Data, can only be interpreted by the MCS. Since the GSC is the bus master, it never needs permission to send data, and will disregard that type of message. When the MCS receives the Command-to-Downlink-Data message, s interrupt handler sets the "clear to send flag," so the transmit software will know it is allowed to send a data block.

The second type of message, the data message, is the most complicated. It contains a byte counter, data and a checksum. The objective is for the interrupt handler to store the data in one of two receive buffers. First, the interrupt handler must determine which buffer to use. It makes this decision based on the status of the buffer's flag and semaphore. The flag indicates if the buffer is full, and the semaphore indicates if the buffer is busy. Figure A-5 gives a visual interpretation of the buffers and flags. If both buffers are full, then the routine will overwrite whichever buffer is not busy. A both-buffers-full situation should never arise, because a buffer is emptied by the main processing unit once every major frame, and a new buffer is received by the telemetry board once every major

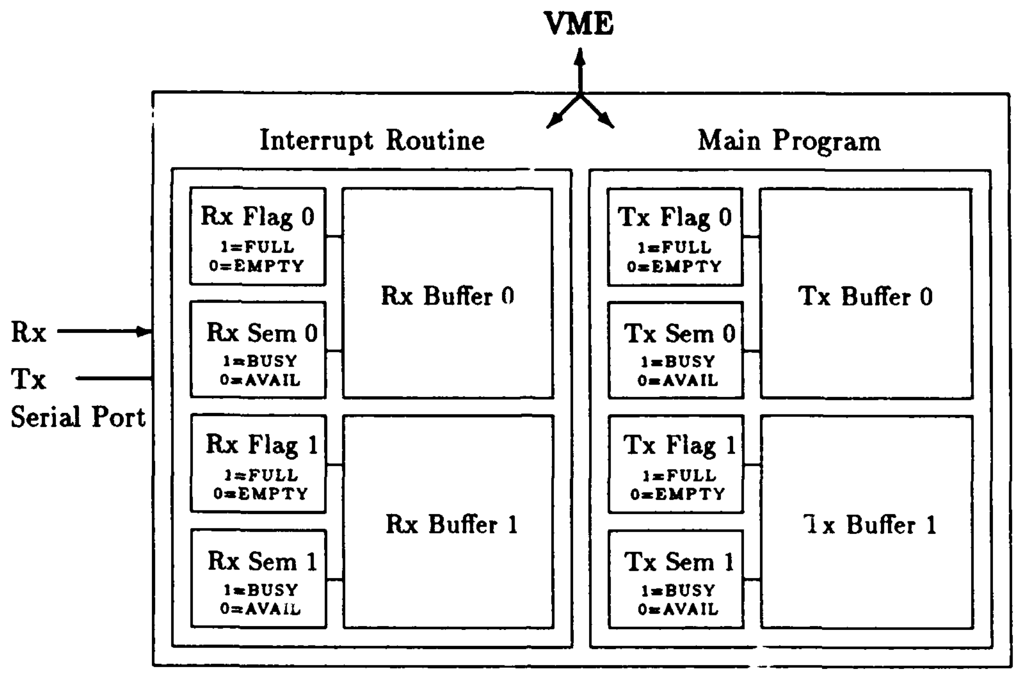

Figure A-5: **Telemetry Board**

frame. As long as the main processing unit and the telemetry board remain synchronized, the current buffer will be emptied before new data arrives. Similarly, both semaphores should never be set at the same time, and this error case will also result in one of the buffers being overwritten. When storing a data message, the interrupt handler maintains a count of how many bytes have been received and how many more are expected. When the specified number of bytes have arrived, the interrupt handler compares the transmitted checksum with its own calculated checksum. If they match, then the interrupt handler sets the buffer's full flag, indicating the buffer has good data. Once this flag is set, it is the main processing unit's responsibility to read the data. If the checksums do not match, then the buffer's full flag is not set, and the main processing does not waste time reading bad data.

The third type of message, the status message, provides two important pieces of information; it indicates whether the previous data transmission was successful and also whether the MCS needs to downlink. When the target board receives this message its interrupt handler interprets the information and reacts accordingly. If the previous data transmission was successful, then the interrupt handler will set that data buffer flag to empty, indicating to the main processing unit that the buffer is ready for new data. However, if the transmission was not successful, then the interrupt handler will not clear the buffer flag, indicating to the transmit software that the data needs to be sent again. The GSC interrupt routine has the additional task of decoding the downlink section of the status message. If the MCS requests to downlink, then the interrupt handler will set the downlink flag. This flag alerts the transmit software that the MCS needs to downlink data.

One additional duty for the interrupt handler is to transmit a status byte in response to a data message. Every time a telemetry board receives a data block, it should immediately respond with a status byte indicating whether or not the checksums matched. The MCS status byte also indicates whether or not it needs to downlink.

A.3 Transmitting

The second section of the telemetry board software controls all the data message transmissions. The messages are stored on the telemetry board in one of two transmit buffers. These buffers are constantly filled by the controller processor unit, transparent to the telemetry software. Similar to the receive setup, two flags and two semaphores control access to the buffers. The transmit routine polls the flags, waiting for an indication that a buffer has just been filled. Although there should never be two full transmit buffers, if this error case does arise, the telemetry board will recover by transmitting one of the buffers, and clearing the other buffer. By clearing the other buffer, the error case will not affect subsequent transmissions. While transmitting the data, the routine sets the corresponding semaphore to "busy," so the main processing unit will know not to fill it with new data. After the message has been transmitted, the routine clears the corresponding flag and semaphore, so the main processing unit will know the buffer is available.

The MCS and GSC transmission routines use the same priority scheme to send messages. However, the MCS requires one more step than the GSC: it must receive permission from the GSC to transmit data. When the MCS needs to downlink, it encodes the request into its return status byte. The GSC stores this request, but only commands the MCS to transmit if there is nothing of higher priority to uplink. New data messages or a retry of a previous message both have higher priority than the MCS downlink. The MCS software loops until permission is received, then begins to execute its send routine.

All interaction with the DUART for data transmissions is handled by a subroutine called *transmit*. It feeds the messages byte by byte, and monitors the transmit ready flag. Before calling this subroutine, the current message must be stored in one of the transmit buffers in the appropriate message format. This format is shown in Figure A-4. The "transmit" subroutine also has the responsibility of calculating and sending the checksum.

A.4 Resynchronization

The purpose of the resynchronization software is to ensure message's are synchronized between the two telemetry boards. If synchronization is not maintained, then the boards could reach a deadlock state, where neither is transmitting because they are both waiting for the other to respond. To prevent deadlock, each board is only permitted to wait a fixed amount of time for the other's response. If the waiting time, called deadspace time, exceeds a predefined value, then the board assumes an error condition, and resets itself. The transmit software is responsible for detecting and correcting all synchronization problems, aided by two subroutines, *wait for status* and *deadspace reset*. The *wait for status* subroutine ensures that status has been returned from the receiver of a data message. This routine will only wait a fixed amount of time for this byte. If the status is not returned, then the sender of the message will assume an error state and execute the other subroutine, *deadspace reset*. This subroutine resets the semaphores and counters so the board will stop waiting for the status byte and begin execution of a new message. Similar to the sender, the receiver of a message also counts the amount of deadspace time. If the sender has not transmitted anything for a predefined amount of time, then the receiver knows there is a problem and calls *deadspace reset*.

The GSC's deadspace reset routine requires extra code to ensure the MCS is reset first. Since the GSC initiates the protocol sequence, the MCS must be ready to receive the data message, or it may miss one or more bytes. To ensure the MCS is ready, the GSC remains idle for a fixed amount of time in its deadspace reset routine, giving the MCS ample time to reset itself. After this idle period, the GSC performs its own reset and returns to sending messages.

As an example of how the resynchronization actually works, suppose that the MCS misses a byte, or receives an incorrect number of bytes. It will then keep waiting for more data and will not return any status. Consequently, the GSC will not receive a status byte and will also be waiting.

The two boards are deadlocked because each is waiting for the other to respond. While it is waiting, the GSC will monitor the amount of deadspace time until it exceeds the predefined value. Then it will enable its *deadspace reset* routine and clear the appropriate flags, and counters. While the GSC is not sending, the MCS transmit routine counts the amount of deadspace time. If this time continues beyond a fixed value, the MCS telemetry board will also interpret the deadspace as an error condition and also reset itself. The MCS should complete its reset routine first and then wait to receive new data. After the GSC completes its reset routine, it will again start sending data messages, and the two systems should be resynchronized.

Acronyms

A/D	Analog-to-Digital
AMCAD	Advanced Multiprocessor Control Architecture Development Project
DUART	Dual Asynchronous Receiver Transmitter
GDCIN	GSC Data Collection Input
GDCOUT	GSC Data Collection Output
GSC	Ground Station Core
MCS	Multiprocessor Control System
MDCIN	MCS Data Collection Input
MDCOUT	MCS Data Collection Output
PWM	Pulse Width Modulation
RTMOS	Real-Time Multiprocessor Operating System
SC	Signal Conditioner
TCB	Task Control Block
TELIN	Telemetry Input
TELOUT	Telemetry Output
URV	Unmanned Research Vehicle

U.S. Government Printing Office 548-076