General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

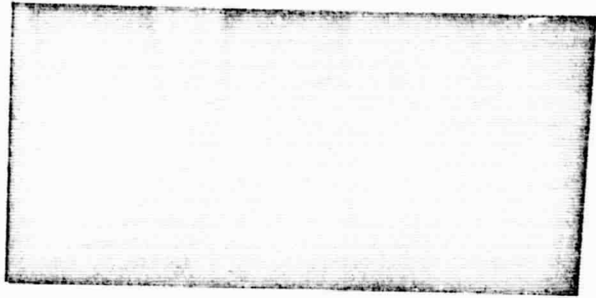
- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)

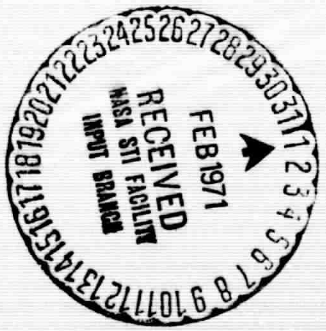


FF 416 DEC 69

N71-15426
(ACCESSION NUMBER) (THRU)
38 G3
(PAGES) (CODE)
CR 114802 08
(NASA CR OR TMX OR AD NUMBER) (CATEGORY)
FACILITY FORM 602

LOGICON

los angeles / san diego / washington, d.c.



CR 114802

Advanced Software Techniques for
Space Shuttle Data Management System

First Interim Report

Contract No. NAS 9-11225

November 20, 1970

Prepared by:

Michael D. Richter
Richard C. Rountree
Raymond J. Rubey
Steven A. Vere

TABLE OF CONTENTS

1. Introduction 1

2. Functional Analysis 2
   2.1 Inertial Measurement Unit 3
   2.2 Unitized Pointing Platform 6
   2.3 Displays 7
   2.4 Trackers 9
   2.5 Landing Aids 12
   2.6 Primary Propulsion Subsystems 14
   2.7 Reaction Control Subsystem 15

3. Computer Configuration Analysis 16
   3.1 Central Computer Facility Configurations 17
   3.2 Executive Design 19
   3.3 Architectural Aspects Study 24

1. INTRODUCTION

This interim report describes the activities performed during the first third of the study of Advanced Software Techniques for the Space Shuttle Data Management System. The results of two tasks are reviewed in this report. The first task is to identify and analyze the functions to be accomplished by the Space Shuttle onboard computing system and estimate their probable computational impact. The activities to be performed in accomplishing this task are complete and the results are presented in Section 2. The second task is to identify the advantages and disadvantages of alternative computer configurations and architectures for the Space Shuttle application so that meaningful hardware/software tradeoffs can be made. The preliminary results of this task are presented in Section 3. It is anticipated that the hardware/software tradeoff activities will continue through the remainder of the study. Other tasks to be performed during this study but not discussed in this report are to investigate the application of higher order programming languages to Space Shuttle software development and to analyze the support software and hardware required for the onboard software's development and verification.

2. FUNCTIONAL ANALYSIS

The analysis of the functions to be performed by the Space Shuttle computer system has been necessary to establish the desired characteristics of that computer system and the computation load to be imposed on it. The computational load can be qualitatively estimated in terms of the computer memory required, the number of instructions that must be executed in a given interval, and the input/output rates that must be maintained. Also important are factors relating to the computational tasks: their relative priorities, their periodicity, the amount of intertask communication, and the number and attributes of routines shared between tasks. As this section will describe, it has not been possible to establish firm and detailed functional requirements at this stage of the Space Shuttle development. Rather, the functional analysis that has been performed has indicated the rough order of magnitude of computational requirements and operating environment for the maximum Space Shuttle avionics configuration. It is apparent that the computational load ranges from one close to that of Apollo to one that large ground-based computers of today would have difficulty in supporting. This range of loads indicates that further study of the hardware interfacing with the computer system and of the functions to be performed by the computer system in supporting this hardware is required to bring the computational requirements within reasonable bounds.

The following avionics subsystems have been examined:

- Inertial Measurement Unit
- Unitized Pointing Platform
- Displays
- Trackers
- Landing Aids
- Primary Propulsion
- Reaction Control

Each of these subsystems is described in the following paragraphs of this section. The input/output requirements associated with each have been given primary emphasis in this study because a proper understanding of these requirements is essential to the computational functions that must be performed. The principal source of information about the avionics subsystem hardware and functions has been the Preliminary Measurement and Stimuli List. Where the documentation is incomplete or inconsistent, assumptions have been made as to the most probable hardware configuration.

A major computational function not yet analyzed in detail is the malfunction (or error) diagnosis, circumvention, and system reconfiguration necessary to meet

the fail-operational/fail-operational/fail-safe requirement. A preliminary survey of this function indicates that the software needed to implement this requirement could double the computational load, while the far greater number of possible sequences of program execution introduced by this software could increase by an order of magnitude the time and cost required for verification and validation activities.

2.1 Inertial Measurement Unit (IMU)

The inertial measurement unit is the most important avionics subsystem for the guidance and navigation function. Two main alternatives exist, each having different computational requirements. The first is a gimbaled IMU such as that used in the Apollo Primary Guidance. Navigation and Control System; the second is a strapdown IMU that is a development of the type of system employed in the Lunar Module Abort Guidance System. A review of documentation and study activities has indicated that for the maximum Space Shuttle configuration, both NASA and the Phase B contractors have given primary consideration to a strapdown IMU. A gimbaled platform would likely be the choice, however, if budget and other constraints dictate the choice of an "off-the-shelf" system.

Because a strapdown IMU presents the most difficult computational problems, four different systems were analyzed as to their computational aspects as summarized in Table 2-1. The four systems are:

- Lunar Module Abort Guidance System (LM/AGS)
- Advanced Supersonic Transport (ASST)
- Redundant Sensor System (RSS)
- MIT Dodecahedral Strapdown Inertial Reference Unit (MIT-SIRU)

Table 2-1. Computer Characteristics for Four Strapdown IMU Systems

Characteristic	System			
	LM/AGS	ASST	RSS	MIT-SIRU
Memory capacity (words)	4096	2150	13,130	16,384
Cycle time (μsec)	5.0	4.0	1.75	0.96
Word length (bits)	18	18	24	16
Repertoire (instructions)	27	NA*	43	NA
Computation time (μsec) Add	10	NA	3.5	1.92
Multiply	70	NA	14	5.76
Major cycle (sec)	2	1-2	1	0.5
Minor cycle (msec)	20/40	10-20	40	10

*Not Available

The LM/AGS was a complete guidance system intended to back up the primary system in the event of failure. It was basically reliant on the primary system for initialization; hence memory capacity for such tasks as alignment and calibration was minimized. There was no requirement for failure detection, diagnosis, and correction logic. The technology era represented is 1964-66, although it has been refined to the present time with a peak emphasis at the Apollo 11 lunar landing in 1968.

Computer characteristics shown for the ASST were for an IMU-local computer. It was intended to be supported by a 6800-word main computer. The ASST was an early attempt at the six-redundant-sensor strapdown approach. Hence, memory capacity for failure detection, diagnosis, and correction logic was optimistic. The overall GN&C subsystem configuration more closely resembles that expected for Space Shuttle than that of LM/AGS or RSS, i.e., the ASST treated multi-IMU/computers, navaids, flight profile prediction and steering, etc. The technological era of the ASST work was 1967.

The RSS was a laboratory experiment to demonstrate reliability/performance capabilities in conjunction with failure detection, diagnosis, and correction logic. The large memory capacity shown is based on an exotic alignment scheme and experiment-related driver subroutines and I/O. More than 7000 words may be expunged if these functions are deleted. A preliminary RSS memory capacity estimate was 2400 words. This was made in 1968, subsequent to the ASST venture. However, detailed examination of failure detection, diagnosis, and correction requirements and alignment and calibration requirements (through to the spring of 1970) greatly increased the memory estimate.

The MIT-SIRU employs six gyros and six accelerometers along axes perpendicular to six of the 12 plane faces of a dodecahedron. The SIRU sensors redundantly measure angular rate and acceleration along the six functional axes. A digital computer program computes vehicle angular rate and acceleration from data received from sets of sensors. By comparing the redundantly determined vehicle data, a failed gyro or accelerometer is detected.

Of the systems studied, the MIT-SIRU is most in accord with NASA requirements. For this reason the MIT-SIRU has been reviewed in more detail than the other, less capable systems. In the MIT-SIRU instrument, loops deliver incremental velocity (ΔV) and attitude ($\Delta\theta$) to a dual redundant multiplexer that transmits data and receives control and sampling messages from the digital computer assembly on dual data buses; a serial data transmission format is used. A local processor has been considered part of the overall SIRU system. The ΔV and $\Delta\theta$ are compensated; checked for failures, malfunctions, or out-of-tolerance behavior; and

converted to attitude and velocity information along equivalent triad-axes (X, Y, Z) by the local processor. The velocity and attitude are then forwarded in dual forms to the central computer facility for guidance and navigation calculations.

Each functional axis contains a ternay torque rebalanced gyro and accelerometer, power supply, electronics, and temperature control. Instrument loops deliver the ΔV and $\Delta \theta$ data at a 4800-pulse/second (pps) rate. Matrix processors and algorithms operate at a 100-update/second rate. Velocity and attitude processing is done sequentially in each 10-msec interval. The local processor would be a 16-bit machine. Data rates are estimated from the following assumptions:

- Single-precision words for three velocity and four attitude (quaternion) components at a 100-Hz rate imply 11,200 bits/sec (bps) during powered flight. Attitude-only during coast yields 6400 bps.

- Failure status of each functional axis is indicated by single bits (go, no-go) to the central computer facility. Display lights are triggered every second, and computer signals are delivered at 500-msec intervals.

The NASA system outlined in the Preliminary Measurement and Stimuli List differs primarily in three areas:

- ΔV and $\Delta \theta$ pulses are sent directly to the GN&C computer

- Different sampling rates are involved

- Additional sensor characteristics (e.g., temperature) are monitored

According to the NASA concept, ΔV and $\Delta \theta$ are delivered to the GSE at a 9600-pps rate for tests. In all other phases they are accumulated in the guidance, navigation, and control computer memory and then monitored at 1 sps. This implies that during the test phases the 12 sensor loops load the data bus at a 115,200-bps rate, assuming 1 bit/pulse. Strapdown inaccuracies become prohibitive when attitude algorithms are not updated at intervals in the range of 10-50 msec.

The requirement to monitor IMU status will increase these data rates. Temperature measurement accuracies call for 11-bit words at 10 samples per

second (sps), yielding 1320 bps to displays and recorder. Blower and heater on/off signals to the recorder and ground display are indicated at 1 sps.

Several comments are offered regarding Space Shuttle philosophy of fail-operational/fail-operational/fail-safe. The first is that a single MIT-SIRU, as presented, will not adhere to the failure criterion. Only two like sensors can fail with the current software. Internal monitoring of the sensors (e.g., gyro wheel speed) will permit three failures. However, if both of the multiplex units fail, the criterion is again not satisifed. Four multiplex units in conjunction with the two sets of six sensors will satisfy the criterion. Catastrophic failure considerations would make two MIT-SIRUs mandatory.

2.2 Unitized Pointing Platform

The unitized pointing platform consists of a mechanical base with two degrees of freedom and a set of three sensors: star tracker, sun sensor, and horizon scanner. Commands from the computing system drive the mechanical axes of the assembly, moving the reference for the sensors. No other source of navigation or alignment information is indicated in the available data, so that it is assumed that star/horizon measurements will be employed for all location data except when one or more tracker is functional. It is assumed that IMU stability is sufficient to obviate the need for use of the platform during atmospheric flight. Initial alignment of the IMU will require stabilization of the spacecraft, acquisition of the sun and the horizon by the sensors, driving the platform by the computer to put each of at least two reference stars sequentially in the field of view of the tracker, and processing of the tracking error signals, platform azimuth and elevation, and spacecraft attitude (from the IMU) for each sighting. Confirmation of IMU alignment is obtained by driving the platform to at least one additional reference star and verifying its location in the tracker's field.

Occasional star sightings will be made under computer control to maintain alignment, and the altitude of a sequence of reference stars above the horizon provides position data. In either case, the computing system drives the platform to the nominal position and reads the sensor error signals. Operationally, one or two such points may be taken every 10 minutes as other tasks permit, so that the residual errors are minimized. Each platform angle covers a range of ±30 degrees, and 16-bit quantization seems desirable. The horizon scanner provides only elevation deviation, which, depending upon the field of view of the tracker, may be quantized with up to 14 bits. Similarly, azimuth and elevation deviations from the sun sensor may be quantized to 14 bits each; practical arguments suggest that 16 bits will be used for each datum. The scanner will provide periodic data, of the order of 10 times per second. The sun sensor and

star tracker have data available continuously, but the former will be sampled up to 10 times per second and the latter need be read only once per star. The above rates are marginal to obtain 5-arc-second accuracy with a residual attitude rate of 0.01 degree per second.

In the absence of utilization data, it is reasonable to hypothesize that the two platforms will be alternated, and that while the sighting frequency may be doubled, the data rates during a sighting will be as given above. If the field of view of the tracker is assumed to be 5 degrees in each axis, 16-bit quantization of its data pair appears more than adequate, yielding less than 0.6-arc-second error. There appears to be no requirement for commanding the platform repetitively once alignment has been obtained; that is, given an inertial reference, a single pair of platform commands will drive the sensors to the desired attitude through a platform servo loop. If the reliability or other cost of such a loop is unacceptable, it will be necessary to generate incremental commands with computer-derived damping to drive the platform to the desired position. In that event, platform angles may have to be both read and commanded at up to 30 times per second to obtain the desired stability, and it may be preferable to use a platform rate command rather than an angle command as the interface. The analogy between these functions and those of the digital autopilot may be strong enough to allow some common usage of routines.

In addition to the normal monitor and command functions, calibration is indicated for the horizon scanner and will probably be required for the other sensors as well. Without data on the mechanisms to be employed, one may only estimate that a few hundred words of program may be needed for each sensor, and that the star tracker and sun sensor would be calibrated once for each set of measurements (perhaps once per 10 minutes of use) and the horizon scanner will require collection of data over several scans (perhaps 10) in the same period.

2.3 Displays

The definition of the display hardware is the least complete of any of the avionics subsystems and, as outlined below, the range of computational requirements is several orders of magnitude. Several significant hardware tradeoffs for the displays are apparent as regards the extent to which analog hardware can be used to reduce the computing load to reasonable proportions. To illustrate this, the displays needed for the dynamic displays which would be required for the blind landing phase have been considered.

One display for blind landing is a video image of the runway as it would appear through the window. From AWAILS or from ILS and the altitude radar, the

computing system determines the vector in inertial coordinates to the end of the runway; trivially, this may be rotated into the vehicle frame. The magnitude of the vector (distance to end of runway) establishes the scale of the image, while the pitch, roll, and yaw angles control the display perspective. Several analog mechanizations exist to provide a CRT image or a projected slide using these data with frequent update. The maximum load they would impose on the computing system would be four single-precision words sent 30 times per second.

The cost of the analog hardware may be considered excessive in dollars, pounds, failure effect, or other significant measure. In that event, the burden may be transferred to the computer, which would provide intensity information for each point of the frame. Even if refreshment of the image were accomplished in hardware, the load would not be eased on an animated display, since the rotating coordinate system of the example requires major processing. We may assume that only a single intensity level is required for the line image to be generated, but we must generate 50,000 to 150,000 bits per frame. To prevent disturbing flicker or erratic motion, the update rate for the display will be of the order of 30 frames per second, yielding a data rate for the one display of the order of 3 million bits per second.

From the point of view of the software, it is probably easiest to locate the image points in the following way. For each frame, each line is represented as a segment in the form $Y = AX + B$, where X and Y are Cartesian coordinates on the display and A, B, and the termini of the segment are computed from the orientation. Each display line establishes a value of X, and each solution represents a point to be displayed. Assuming six such lines in the image, six solutions must be sought, each involving a product, a sum, and at least two tests. With each solution found, allowance is made for the finite width of the display line by providing a number of consecutive dots equal to the inverse of A for all A less than 1. (That algorithm allows the effective brightness of the line to increase by 40% as it approaches the perpendicular to the scan line, but it is the simplest implementation.) The segment is truncated as required.

For each display frame, the computer must determine six sets of four segment parameters and one inversion. For each of the 400 to 500 lines in a frame, it must find the solution (one multiply, one add), determine the left and right termini of the line segment (one add each), test each terminus for inclusion in the display (four or eight compares), and assemble a sequence of binary words with each bit corresponding to a point on the scan line. The first such set is stored; each of the five successors is OR'ed with the first

on a word-by-word basis to establish the composite display. Typically 15 such words will be required to depict a scan line, suggesting a composite requirement for each line in excess of 6 multiplies, 18 adds, 24 compares, and 15 logical ORs. The available time is of the order of 30 msec divided by 400 lines, or 75 μsec. Assuming that 10 adds require the same time as one multiply, the arithmetic operations alone suggest a maximum add time of less than 1 μsec for final computation.

Note that the four items required for command to the analog system are essentially available in the computing system without further processing; in the all-digital configuration they are supplemented by six projections in three-dimensional space. The four parameters of each segment must be determined for each frame, suggesting that a typical CPU with 0.5 μsec add time would be hard pressed to handle the animated display if dedicated to it full time. Recognizing that only a single, simple display of its type has been considered, one sees easily the impact on system configuration of display requirements. The requirement to support colored displays would further increase the computing load. Simple and multiparameter trend analysis may be added, further to increase the computing load for related displays.

2.4 Trackers

Three tracking subsystems have been reviewed: the docking laser, the rendezvous radar, and the infrared tracker. No interaction between these subsystems has been described, although it is likely that it will exist and that it will have significant computational impact. The docking laser has no equivalent existing projects and therefore its potential computational impact has been given the most consideration.

2.4.1 Docking Laser

The docking laser collects data for the determination of target attitude to assist in automatic docking. It transmits a beam to the passive base, receives reflections from each of the three targets on the docking ring, and determines the range and some relative attitude data from analysis of the detected signal. Dual-tone laser modulation allows operation over a range of 0.25 statute mile without ambiguity, and the system is capable of accuracy to a fraction of 13 feet (equal to the wavelength of the high tone). Incoming light is collected by a Cassegrain telescope and passed through a filter to an image dissector. Two degrees of freedom are scanned electronically within the image dissector upon command from the computing system; either a positional command or a discrete may be employed depending upon the sophistication of the scan control and monitor electronics.

Two cases for scan control and monitor electronics have been considered: a sophisticated processor internal to the laser system and a simple interface package requiring the central computer facility to do most of the processing. Sophisticated scan control and monitor electronics will provide a set of scan patterns depending upon the quality of the data collected in the scan itself. With such a package, the system inputs reduce to a single discrete to initiate scanning in both directions, with the actual pattern under control of the laser electronics. Scan termination would be commanded by removal of the discrete. Feedback would be required from the range computer to the electronics to signal the presence of a return. The computing system for so elaborate an internal scan-control package would do only the processing required for guidance purposes. The internal evidence favoring such a configuration as assumed here includes the identification and correction within the laser system of glare areas and the simplicity of the indicated interfaces with the computer.

The simple interface package would accomplish format conversion and little more. For this alternative, the computing system must maintain a map in memory of the pattern of returns, command each displacement of the image dissector aperture separately, and maintain detailed control of the scan pattern. The suggested command rates would be up to 10^6 displacements of the effective aperture per second in each of the two directions; more probable rates will be 10^4 during the most active scanning phases (e.g., acquisition of target) and less than 10^3 for the others.

The azimuth and elevation of the target returns are provided to the computing system for interpretation. A separate computer detects range through phase measurement and transmits two types of data:

- Low tone range, apparently whole cycles of the high frequency between transmission and return, hence scaled as integers from 0 to 200

- High tone range, presumably multiples of some fraction of a wavelength of the high frequency, probably requiring four or five bits in encoding.

Each scan angle may be encoded with up to eight bits, although a reasonable design of the optics might allow five bits to suffice, and other considerations might prompt an attempt to use 10. A set of laser data (high and low tones, azimuth and elevation angles) may be available as often as 1,000 times per second with the operating frequencies in use.

There are three targets on the base, so the solution of base position is overdetermined. One computational approach to the utilization of the laser data has been reviewed to determine the possible computational impact. In this approach three major routines are required for determination of docking data: a dynamic model of the base, a Kalman filter, and a coordinate converter. The last adapts input data to the requirements of the model, presumably resolving each input set into inertial position of its target. The laser may provide 1,000 sets per second, and the radar 10 sets per second. Each set requires development of two sine-cosine pairs, four multiplies, and two additions.

The measured data are compared with the set from the six-degree-of-freedom model. Since the Kalman filter to be applied to the derived error is the most complex routine in the set, it may be desirable to rotate the coordinate system into a frame suitable for estimation of errors. Note that the base may be executing attitude maneuvers (including those associated with holding attitude) during the period of measurement, and that no external source is available to identify firing of the base reaction control system. Thus, the Kalman filter will have to identify a probable firing for the model.

The Kalman filter derives best estimates of target state and attitude and of errors in the data. Its use requires the development and inversion of a large matrix (depending upon the number and size of the error sources), yet it must be amenable to rapid cycling so that docking may be performed in real time. There exist three parameters of position error and three of velocity for each of the four targets under consideration (including the transponder), plus two or more for each degree of freedom in data collection. This suggests that the matrix for inversion will be of the order of 36 by 36 elements. The computational load resulting from the inversion of such a large matrix every millisecond is beyond the capability of a foreseeable onboard computer system. Both the docking laser hardware and the method by which its output is utilized clearly must be more fully analyzed so that a practical system can be developed.

2.4.2 Rendezvous Radar

The rendezvous radar is also referred to in the available documentation as the microwave tracker and the microwave rendezvous tracker radar. The 9-GHz beam is transmitted and received through a diplexed, phased array, and the output data are the commanded angles and the derived range and range rate. As in the docking laser, the question of drive for tracking is not

resolved in the available material; search patterns may be generated internally, or they may be obtained through the computing system. It appears unlikely that skin-track capability is incorporated in the rendezvous radar; if this is correct, then rendezvous with a passive base must be effected without rendezvous radar data, at least until very close range. Although the indicated data rate is 1 sps (each sample consisting of two angles, range, and range rate), it seems likely that up to 10 times that rate might be desired in conjunction with the automatic docking mode. Each datum should be encoded with 16 bits. If steering of the beam is to be a computing system task, two words of 16 bits each will be required (azimuth and elevation to the base). The finer steering required to keep the beam locked onto the target is a normal function of the demodulation electronics, so that the steering command will not require reiteration while the "data good" discrete is supplied by the rendezvous radar.

2.4.3 Infrared Rendezvous Tracker

The infrared rendezvous tracker detects the thermal radiation of the base to determine polar coordinates within its field of view (5-degree full-cone angle about the longitudinal axis). Scanning is indicated as entirely under internal control. Data should be available at least once per second, each set consisting of the azimuth and elevation of the base, with each angle encoded to no more than eight bits (four would suffice for a small telescope even using a cooled detector). The signal data converter is shown as resolving the polar data into azimuth and elevation; no advantage is evident for this configuration over performing the conversion in the computing system. Complementing the usual monitor for system performance is the set required for control and monitoring of detector temperature. Depending upon detector design, it may be possible to use the infrared rendezvous tracker with the sun in the field or reflected by the base. It is possible that sun-avoidance logic for the scan pattern may be required, suggesting two eight-bit words per second from the computing system to the scan control when the sun is in the field of view.

2.5 Landing Aids

Five landing aids have been reviewed: altitude radar, ATC transponder, VOR TACAN, instrument landing system, and all-weather automatic instrument landing system. The last three systems are complementary in principle but will have significant periods of combined operation where their data will be supplementary.

2.5.1 Altitude Radar

This radar determines height above the local terrain from the transit time of a signal at 9 GHz. Since terrain variation will be a significant factor in the received data, smoothing will be required by the computing system. It seems likely that 16 bits will suffice for the data, and that a nominal rate of the order of 100 msec between samples will provide rapid response and effective smoothing. One possibility is that landing at the nominal site would call for removal of known terrain variation from the data, either complementing or replacing smoothing. Since the vehicle must be capable of landing at unplanned sites, such a feature for the nominal seems to be of little advantage, while it might commit a significant portion of memory.

2.5.2 ATC Transponder

The air traffic control transponder receives its input at 1030 MHz from the ground, converts it to 1090 MHz, modulates it with selected guidance data, and transmits the resultant. Normal functions of monitor and self-check are required, but the interface for guidance data must also be supplied. No interface between the ATC transponder and the guidance system is shown in the Preliminary Measurement and Stimuli List. It is probable that the following data should be required no more than once per second:

- Reference time (GMT)
- Mean altitude above terrain (unweighted average of altitude radar data)
- Inertial state vector (referenced to the rotating earth)
- System status code (synthesized from individual status data).

Other data available in the landing phase would appear to be of little value to those receiving the transponder signal. The computing system interface is independent of the operating mode of the transponder during landing phases. The resulting data rate is of the order of nine single-procision words per second, all drawn from existing data and requiring a minimum of computation.

2.5.3 VOR TACAN

VOR TACAN is a pair of data sources (omnirange and TACAN) providing inflight data relative to earth-fixed transmitters. Each system employs a

signal from the ground decoded onboard into an identifying tone for the transmitter, a reference bearing, and a variable bearing as a function of vehicle position relative to the antenna. It is assumed that manual identification of the station will be employed; the A/D and digital requirements for automatic operation seem more costly than this supplementary information is worth to the mission. TACAN also includes an active ranging path to its antenna. Particularly because of the short range over which VOR TACAN data are good, eight-bit encoding of each bearing signal and of range would appear sufficient. While data are available essentially continuously, sampling intervals of 1 second are sufficient unless TACAN data are to be used to derive wind velocity. In that case, 12-bit range and bearing data will be needed, and rates of the order of five samples per second may prove useful.

2.5.4 Instrument Landing System (ILS)

The ILS brings the vehicle to the middle marker of the runway through detection of azimuth and elevation from a reference transmitter. A supplementary antenna signals passing of the outer and middle markers. The detected signals of glide slope (elevation) and localizer (azimuth) may be usefully encoded to about eight bits and sampled once per second; the two marker signals should be discretes to the computing system, and require sampling at the same rate but for only the brief period near their expected occurrence. Ambiguities exist in ILS data which may require resolution in the computing system both for display and for automatic landing. Comparison between IMU and ILS data should be sufficient to determine which of the five possible ILS references has been detected and to optimize the landing trajectory.

2.5.5 All-Weather Automatic Instrument Landing System (AWAILS)

AWAILS provides angular interfaces equivalent to those of ILS and supplements them with measurement of range to a transponder on the runway. Again, eight-bit encoding of each of the three data should suffice and one sample per second would appear adequate. AWAILS would be employed only in the final stages of landing.

2.6 Primary Propulsion Subsystems (PPS)

The primary propulsion subsystems employ two separate sets of engines. The main engines are used for the final stage of boost, while subsequent major thrust maneuvers employ the orbit maneuvering subsystem (OMS). Hydrogen fuel is used to cool each active nozzle regeneratively; the oxygen oxidizer drives both turbopumps for the OMS, while preburners drive pumps in the main engines. Helium is used for pressurization of each subsystem.

A reasonable construct for operation of either engine subsystem calls for a set of pre-ignition commands over a period of seconds to minutes, an engine-on signal maintained throughout thrusting, separate gimbal angle commands of indeterminate rate and quantization for either degree of freedom, and a mixture-ratio command. In the absence of data about a specific PPS configuration, the following estimates are made for the design carrying least load on the computing system.

About each axis, each engine may be gimbaled approximately 5 degrees, and quantization to 0.1 degree would be useful. A reasonable interface would allow for up to 10 commands per second per axis, each incrementing or decrementing the gimbal angle by one step. Initiation of thrust may use two discretes (engine sequence start and thrust start), or it may require a sequence of discretes issued on a time base; the most difficult implementation would provide discretes triggered by measurements of PPS parameters under computing system control. Measurement of propellant quantities and flow rates provides the data for computer determination of the desired mixture ratio; the increment/decrement discrete is required no more than once per second. Each incremental signal may require a discrete which causes the controller to drive to the reference (null) position. The normal monitoring and mode-control functions should be assumed.

2.7 Reaction Control Subsystem (RCS)

The reaction control subsystem employs 20 thrust chambers burning hydrogen and oxygen to generate relatively small torques and forces for attitude control and for small velocity increments. Since the propellants are not hypergolic, a high-voltage discharge is required to ignite each pulse; whether a continuous-thrust mode is to be provided is not clear from the documentation. If continuous thrusting is possible, the output of the computing system may be a discrete for each chamber (assuming hard wiring rather than the use of the data bus). In that event, the signal is counted down with a tolerance of the order of 1 to 5 msec. If only a pulsed mode is available, the number of such pulses (up to perhaps 800 per second) must be transferred to the buffer in a local processor for each assembly. Two or three such items are required for each assembly, depending upon whether it includes four or six thrust chambers.

Many configurations may be constructed in which computing system functions are assigned to local hardware and software. The above configuration does not allow for some modes of employing rate gyros, nor is it clearly the preferred type of interface if control surfaces are to be used in atmospheric flight. The specific mechanism by which a command signal actuates the RCS is not clear from the available documentation, and is significant in determining the software requirements.

3. COMPUTER CONFIGURATION ANALYSIS

In addition to a determination of computational requirements from the functional analysis, performance of a hardware/software tradeoff analysis requires an assessment of the software aspects of alternative hard ware configurations. The functional analysis is directed towards the determination of whether a particular computer configuration can solve the problems arising in the Space Shuttle application. The computer configuration analysis is directed towards the selection of the configuration which minimizes the effort and cost required to produce the software.

The baseline computer configuration consists of a central computer facility (CCF) communicating with standard acquisition control and test units (ACT) or digital interface units (DIU) bus as shown in Figure 3-1. The alternatives in this baseline which are being addressed in the computer configuration analysis include the configuration of the CCF, the partitioning of computational tasks between the CCF and the ACT/DIUs, and the architectural features desirable in the CCF and the ACT/DIUs. The study to date has concentrated on factors influencing the choice of a particular CCF. Analysis of a particular CCF configuration requires an understanding of the executive programs which control execution of tasks and an evaluation of the influence particular architectural features have on that configuration's programmability, software maintainability, and ease of software verification and validation. CCF configurations are discussed in Section 3.1, executive designs in Section 3.2, and architectural features in Section 3.3.

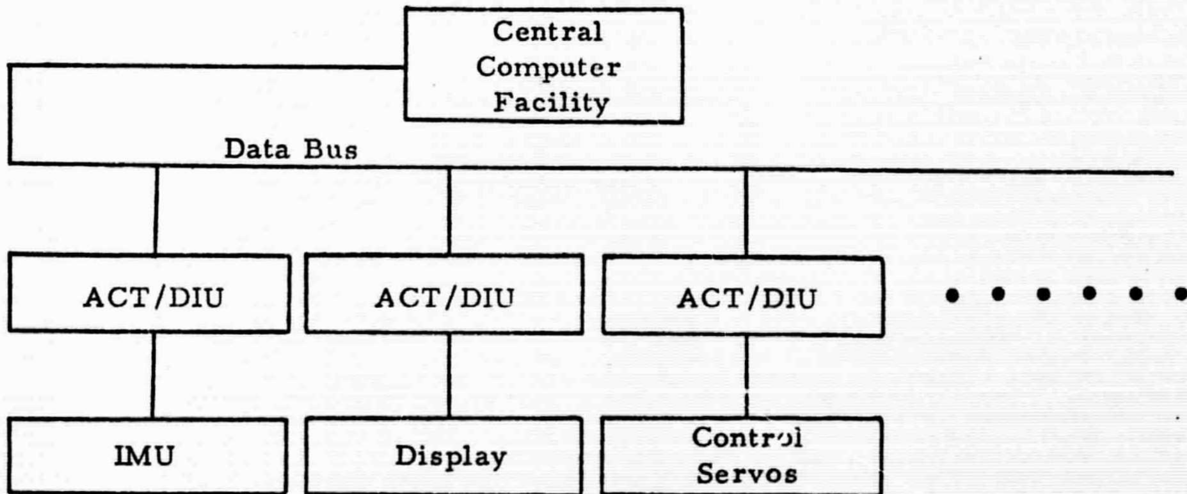


Figure 3-1. Baseline Computer Configuration

3.1 Central Computer Facility Configurations

A simplex organization is clearly the most desirable in terms of programmability and verification and validation ease. However, its disadvantages -- the relatively restricted computational capability it affords and the redundant hardware required to meet the reliability requirements of the Space Shuttle application -- have made it necessary to study multicomputer and multiprocessor organizations. For this study a relatively restricted definition of multicomputer and multiprocessor has been established. As shown in Figure 3-2, a multicomputer configuration has two or more central processing units (CPUs), memories, and input/output processors. A multiprocessor similarly has two or more CPUs, memories, and input/output processors, but communication is by means of direct memory access.

The advantages of a multicomputer organization are several. No hardware or software scheme is required to lock out a processor from a memory. During debugging, errors in a task on one processor cannot disrupt tasks on others. The isolation of memory units may make verification and validation easier. Memory access time is not degraded by competition between processors. The disadvantages of the multicomputer organization are that the I/O processor must be called to transfer data between memory units for tasks running on different processors. Duplicate utility routines, executive routines, etc., must exist in each memory unit. It is difficult for processors to share workloads equally: one processor might remain idle while another had a backlog. The only recourse would be to transfer the task routines between memory units using the I/O processors, or to duplicate the routines in each memory.

Among the advantages of the multiprocessor is the fact that common routines, such as executive and utility routines, need not be duplicated. It is not necessary to transfer data or programs between memory units when various tasks share data. Assignment of processors is flexible: if Task A is running on Processor 1 and is interrupted by Task B, which is of higher priority, Processor 2 can take up Task A when it finishes its own current task. Thus the processors can be kept busy and throughput is increased. The disadvantages of such an organization include the need for a hardware or software memory lock scheme to insure orderly use of common data and programs. Memory access time is degraded when processors try to access the same module at the same time. During program testing or operational use there is opportunity for an error in one task to disrupt tasks on other processors. The high flexibility permits a large number of possible system states, making impractical verification and validation and by exhaustive testing alone.

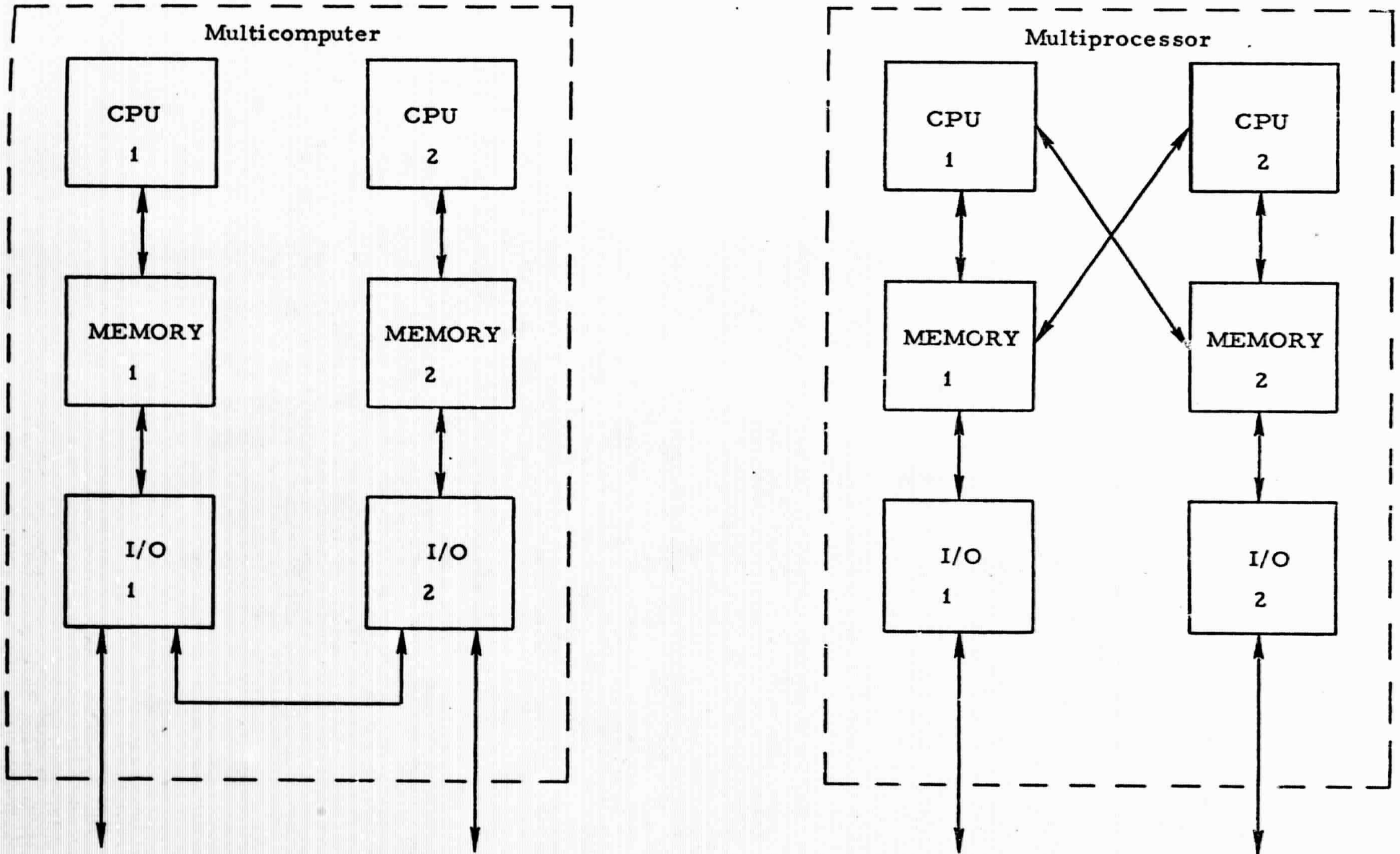


Figure 3-2. Multicomputer and Multiprocessor Configurations

The multicomputer thus tends to be less efficient but makes verification and validation easier than the multiprocessor. However, if multiprocessor capabilities are not utilized to promote efficiency, the multiprocessor can result in software simpler than that of a corresponding multicomputer. This can be seen by dividing all the software routines of the multicomputer shown in Figure 3-2 into the following four categories:

Job routines (R_J)
Executive routines (R_E)
True I/O routines (R_I)
Intercomputer data transfer routines (R_T)

Assume that there exists the multiprocessor also shown in Figure 3-2 with identical CPUs, memories, and input/output processors as that of the multicomputer and differing only in the additional memory-CPU channels. The addressing of Memory 1 is exactly the same in both systems. If the size of Memory 1 is ℓ_1 and the size of Memory 2 is ℓ_2, in the multiprocessor configuration the addresses in Memory 2 range from $\ell_1 + 1$ to $\ell_1 + \ell_2$. Routines R_{J1}, R_{E1}, and R_{I1} are transferred unchanged from the multicomputer to multiprocessor Memory 1. The address references in R_{J2}, R_{E2}, and R_{I2} are modified to fall in the range ℓ_1 to $\ell_1 + \ell_2$, giving R'_{J2}, R'_{E2}, and R'_{I2}, which then go into multiprocessor Memory 2. The intercomputer routines R_{T1} and R_{T2} are replaced by a simpler set of memory copy instructions R_{C1} and R_{C2}.

Thus the software changes required to go from the multicomputer to the multiprocessor involve simplifications if no attempt is made to take advantage of multiprocessor capabilities to increase efficiency. Intertask communication in a multiprocessor can be accomplished in a way similar to a simplex computer, avoiding input/output complexities that can occur in a multicomputer configuration.

Thus two factors are important in the selection of a central computer facility configuration: the extent of the intertask communication and the design of the executive to control the sequencing of task execution. The first factor is addressed in the functional analysis phase of the current study (Section 2). The alternatives in executive design are discussed in Section 3.2 following.

3.2 Executive Design

The primary function of the executive program is control of scheduling, initialization, execution, and termination of tasks. The executive also

controls input/output operations, provides a mechanism for reconfiguration of the computer system when hardware failures occur, and furnishes the utility subroutines common to several tasks. The functional analysis of the avionics subsystems has indicated that the Space Shuttle computational problem inherently involves operation of the central computer facility in a multiprogramming mode. For example, the calculations of vehicle attitude and position, the reduction of information from the docking laser, the monitoring of subsystem temperatures and voltages, and the generation and refreshment of displays are tasks which all must be accomplished within timing restrictions which are largely independent of the requests for performance of these tasks. This requires the establishment of a priority structure by which the executive determines the tasks that should be performed at each interval of time. A complication arises when the time that a task can wait before its processing can begin is shorter than the length of time that another task takes to perform. This would require that the first task interrupt the second if the timing requirements were to be satisfied or that the second task be broken into shorter subtasks. Regardless of the approach taken, the situation occurs in which two tasks are simultaneously in a state where their execution has begun but is not yet complete. This concurrent execution of two or more tasks is the essence of multiprogramming.

If there were no hardware limitations and intertask communication were at a minimum, a simple solution to the problem introduced by multiprogramming would be to dedicate a computer to each task. For the number of tasks in the Space Shuttle application this is clearly impractical. Therefore, the executive program must do the scheduling such that tasks can be executed in parallel. The problem of developing an executive to accomplish such multiprogrammed scheduling is no more difficult and may even be easier for a simplex configuration than for the multiprocessor or multicomputer configurations in which the number of processors or computers is significantly less than the number of tasks.

An important consideration in executive design is the separation of executive functions from the task programming. To the greatest extent possible, the design, coding, and checkout of a task should be independent of the executive actions taken to schedule that task. Similarly, the executive should be independent of the specific tasks so that it is easy to add or delete a task in the set of tasks to be scheduled and executed. If the executive is modular, such changes in the tasks to be scheduled or the attributes of the tasks themselves do not require corresponding changes in the executive. In terms of organization, a modular executive will probably use a priority scheduling scheme. While the past priority schemes have generally

been applicable only to aperiodic tasks, the Space Shuttle computational problem is characterized by a high percentage of tasks that must be executed periodically. Fixed-priority scheduling is not meaningful for periodic tasks. Periodic Task A may not have a subjectively high priority relative to Task B, but when its cycle time arrives it must be executed, even if the execution of B must be suspended in some manner. This can be accomplished by making the priority of a periodic task a step function variable as shown in Figure 3-3. The upward step occurs at the cycle times, and the width of the step is the execution time of the task.

If the task need not be performed on an exact interval, but only regularly at approximately a given interval, the priority can be made some increasing function of the time since last execution as suggested in Figure 3-4. If high-priority aperiodic tasks must be scheduled together with fixed-length periodic tasks, hardware efficiency will be low since large empty time spaces must be left between the processing of the periodic tasks to insure that the high-priority aperiodic tasks will never be delayed very long. If a dynamic priority scheme is used and "periodic" task routines are written to allow for variable-length tasks, these empty time spaces are obviated. When the aperiodic tasks occur, they can squeeze in between the periodic tasks, and empty time spaces need not be provided a priori. The tradeoff is the software complexity introduced by allowing variable-length periodic tasks and the dynamic priority scheme versus the hardware efficiency resulting from an elimination of need for empty time spaces.

In a multicomputer or multiprocessor configuration, decisions must be made as regards the residency of the executive and the permitted flexibility in task allocation among processors. These decisions affect the efficiency of the executive, the difficulty of its development, and the problems encountered in verification and validation. The options as to executive residency are that:

1a) The executive functions are (or can be) performed by each processor

1b) The executive functions are performed by only one processor.

The options as to the flexibility of task allocation are that:

2a) All job tasks can be performed by all processors

2b) Job tasks are divided into sets, and each set is allocated to a different processor.

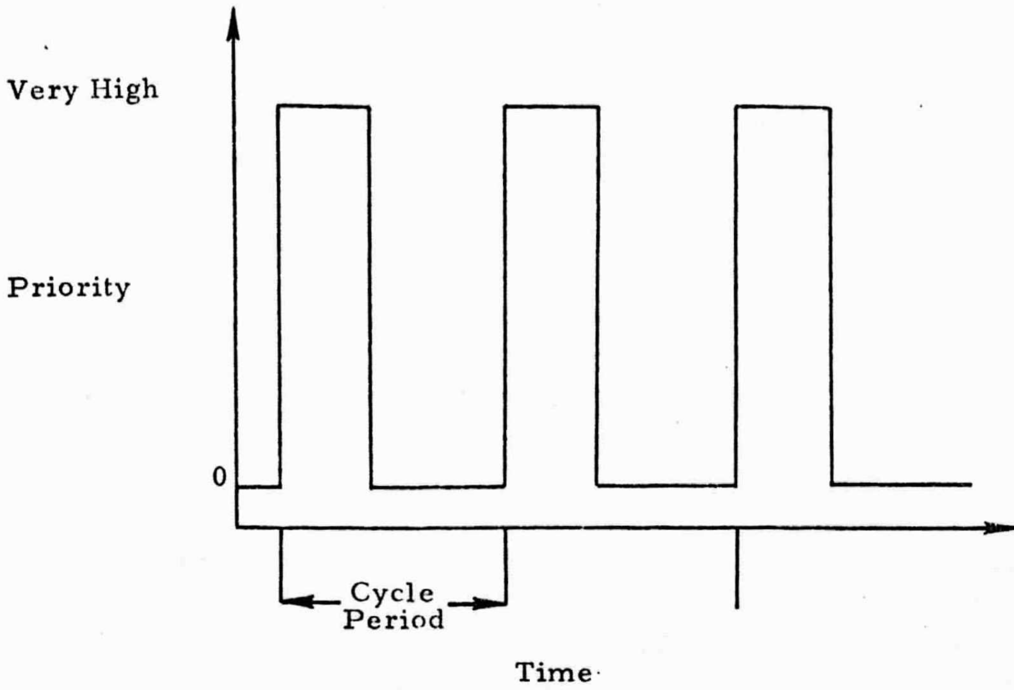


Figure 3-3. Priority of Periodic Tasks

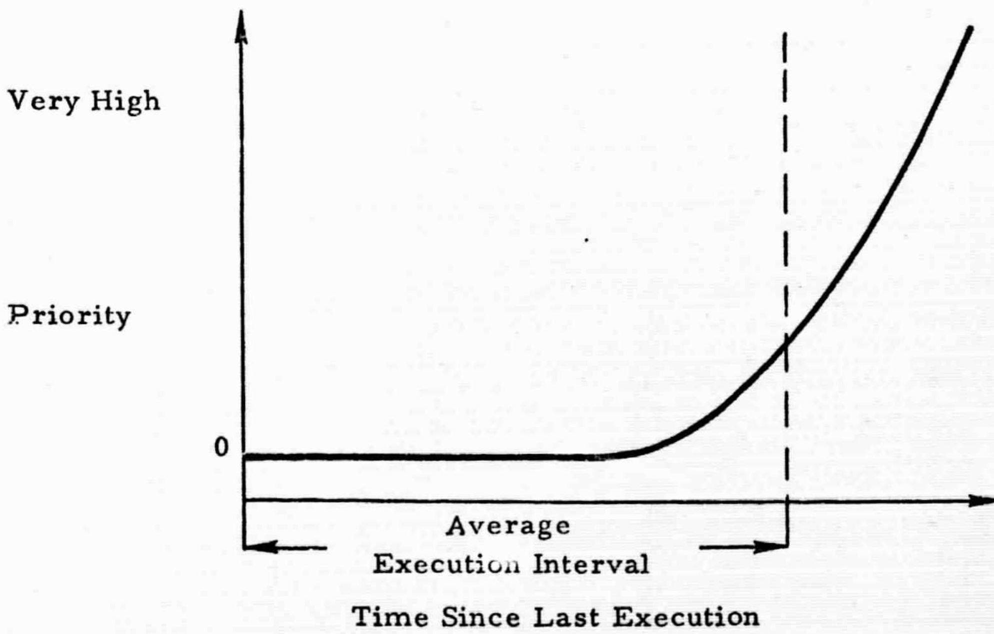


Figure 3-4. Priority of Quasiperiodic Tasks

Three basic choices have been considered for the multiprocessor executive. A multiprocessor floating executive consists of Options 1a/2a. A multiprocessor distributed executive consists of Options 1a/2b, that is, each processor performs the executive functions for its own set of tasks. A multiprocessor dedicated executive corresponds to Options 1b/2a or Options 1b/2b.

In the multiprocessor configuration, Option 1a implies that any processor must be able to interrupt any other processor, while Option 1b requires only that the executive processor be able to interrupt the other processors. Because of the asymmetry implied, 1b is not as desirable in view of the redundancy requirement, but does result in a simpler executive. Option 2a implies that the task scheduler will use a single job queue. Option 2b implies that a separate queue will be used for each processor. For the single-queue case, a processor will never execute a task having lower priority than one in queue, and no processor will ever be idle unless the queue is empty. For the multiqueue case, it is possible for a high-priority task to be in queue for one processor while another processor is executing a task of lower priority. In this case it is also possible for one processor to be completely idle while tasks are still in queue for another.

The choices in multicomputer executives are more limited than those available for multiprocessors. The practical executive options for a multicomputer are 1a/2b and 1b/2b.

Option 2a is impractical for a multicomputer, since it would require that routines be transferred between the memories of the various computers. Option 1b would require more intercomputer I/O, and also might imply lower hardware efficiency if the total amount of executive tasks were not enough to keep a processor busy. It is also hard to imagine how one computer could handle I/O for the others, and so a dedicated executive multicomputer system would be primarily a dedicated scheduler system.

Detailed study of the alternatives in executive design and their advantages and disadvantages requires an examination of the types of architectural features that can reduce or eliminate common problems. Some of the architectural aspects are discussed in the following section.

3.3 Architectural Aspects Study

Software criteria such as computational capability, programmability, and ease of verification and validation for a particular computer system depend on both the gross configuration (i.e., simplex, multicomputer, or multiprocessor) and the specific architectural aspects of the system. For example, a multicomputer configuration with all of the desirable architectural aspects such as floating-point arithmetic, memory lock, and adequate word length might be more desirable in terms of the above software criteria than a simplex computer lacking these features.

The architectural features of proposed configurations can be divided into two classes:

1) Internal features are those which affect the programming of a single task. They will influence both job task and executive functions.

2) External features are those which affect the programming of interaction between one task and another, or between a task and the external world. They will influence only the executive functions.

Typical internal features are the job instruction set, addressing scheme, word length, microprogramming facility, and internal register system. Internal features are largely configuration-independent. Thus, for example, the utility of a floating-point instruction set is the same for both the simplex and multicomputer configurations. Typical external features are executive or mode instructions, interrupt facility, memory lock features, and I/O features. External features are configuration-dependent; for example, the utility of a memory lock feature would be dependent on whether a multicomputer or a multiprocessor configuration was being considered.

Two external and one internal architectural aspects have been examined in detail. The external ones considered are interrupt mechanisms and memory locks; the internal one is register stacks. Each of these is discussed in the following paragraphs.

3.3.1 Interrupt Mechanisms

An interrupt mechanism, a common feature of real-time control computers, serves to signal to the executive that a request for the processing of a particular task or set of tasks has occurred. If properly designed and used, interrupts can contribute to efficient utilization of a computer's

resources. Interrupts are properly the concern only of the designer of the executive routines and ought to be invisible to the task programmer.

To interrupt a task running on a processor, it is sufficient to store the contents of the instruction register and internal data registers of the processor in memory. A pointer to this information is retained in a list to permit the task to be reloaded and continued at a later time. To make interrupts as quick as possible, it is desirable that the number of internal registers not be too great, since the contents of each one must be stored to effect the interrupt. If an elapsed-time register exists, this too must be stored. The minimum interrupt mechanism is a signal (the interrupt signal) which can cause an immediate branch to some specified instruction after storing the contents of the instruction register. A microprogrammed interrupt procedure could automatically store the contents of the machine's internal registers on receipt of an interrupt signal. For the Space Shuttle application it is highly desirable that enough computer memory exist so that for a given mission phase, all necessary task routines would reside in main memory. Consequently, the only time penalty involved in an interrupt would be the time necessary to store the internal registers. To begin a new task a memory swap with an auxiliary storage device would not be necessary.

Two classes of interrupts can be distinguished for the Space Shuttle application:

1) External interrupts are those initiated by local processors or sensors.

2) Executive interrupts are those generated internally by the executive routines. For example, if a task tries to access a locked section of memory, it would be interrupted and placed on a waiting list until the memory section is unlocked.

The external interrupts would be received (possibly after a poll) by an I/O processor, or in a multiprocessor configuration by the processor performing I/O functions. In a multiprocessor, only the processor with the lowest priority task should be interrupted. It is desirable that the processor to be interrupted be chosen (either by hardware or software means) without interrupting another processor to perform an executive routine, since doing so would increase executive routine overhead.

It has been suggested that the need for interrupts can be eliminated by dividing all tasks into segments, each one so short that it can always be

allowed to finish before starting another. Two disadvantages of this approach are that overall computer response time is degraded and extra effort must be expended in segmenting the program. Further, if changes in program requirements, design, or implementation occur the resegmentation may involve a disproportionate amount of additional effort. It can be seen that a system which attempts to get by without interrupts is less flexible than one which uses them as a programming convenience. Segmentation of tasks and the elimination of interrupts has the advantage that the point where a task is interrupted is predetermined and therefore verification and validation can be simplified.

The extent of the savings in verification and validation becomes apparent from a look at the conditions which are necessary to prove correct program behavior. In the no-interrupt situation it is very probable that it will be impossible to determine the sequence in which an arbitrary pair of independent task segments (say A and B) would be executed. Thus it would be necessary to prove that sequence A-B produced the same result as sequence B-A. In other words, what is required is a demonstration of the commutativity of A and B. For the interrupts-allowed case it is necessary to prove that at any point in the execution of Task A it is possible to execute Task B and vice versa. This is logically identical to proving that A and B can be executed in parallel. Techniques that exist for determining whether two programs can be executed in parallel are directly applicable to demonstrating the mutual interruptibility of two programs. An examination of such techniques and their application to the demonstration of commutativity gives insight in the verification and validation activities required for each case.

In considering the memory locations used by a routine, four sets can be distinguished:

1) $M_1(A)$ is the set of memory locations which are only read by routine A.

2) $M_2(A)$ is the set of memory locations which are first read by A and later written into by A.

3) $M_3(A)$ is the set of memory locations which are only written into by A.

4) $M_4(A)$ is the set of memory locations which are first written into by A and later read by A.

In considering interruptibility of routines, a simpler categorization can be used. Define

$$R(A) = M_1(A) \cup M_2(A) \cup M_4(A)$$
$$W(A) = M_2(A) \cup M_3(A) \cup M_4(A)$$

The R(A) is the set of locations which are read by A. W(A) is the set of locations which are written into by A. If routine A is involved in any I/O activity, the inputs to A must be included in R(A) and the outputs from A must be included in W(A). In deciding if two routines can be executed in parallel, the situation is as shown in Figure 3-5A. A and B are two routines which tentatively have a serial ordering. P represents the remainder of the program. The conditions insuring that A and B can be executed in parallel (as in Figure 3-5B) are:

1) $R(B) \cap W(A) = \varphi$, where φ is the null set. (B does not read any memory locations written into by A.)

2) $R(A) \cap W(B) = \varphi$ (A does not read any memory locations written into by B.)

3) $W(A) \cap W(B) \cap (M_1(P) \cup M_2(P)) = \varphi$ (P does not first read from any locations written into by both A and B.)

Now consider the problem of interrupts. Two routines, A and B, executed in "normal" order can be indicated as in Figure 3-5C. If B can interrupt A, the execution can be indicated as in Figure 3-5D. A_1 and A_2 indicate the two segments of A which are separated by the execution of B, α indicates the interrupt point, and P indicates succeeding routines. To insure that the interrupt cannot cause software failure, the following requirements must be satisfied:

1) $W(A_2) \cap R(B) = \varphi$ (A_2 must not write into any locations which are read by B.)

2) $R(A_2) \cap W(B) = \varphi$ (A_2 must not read any locations which are written into by B.)

3) $W(A_2) \cap W(B) \cap (M_1(P) \cup M_2(P)) = \varphi$ (If A_2 and B write into any common locations, these locations must not be read by any succeeding routines.)

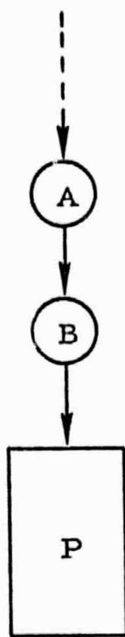


Figure 3-5A

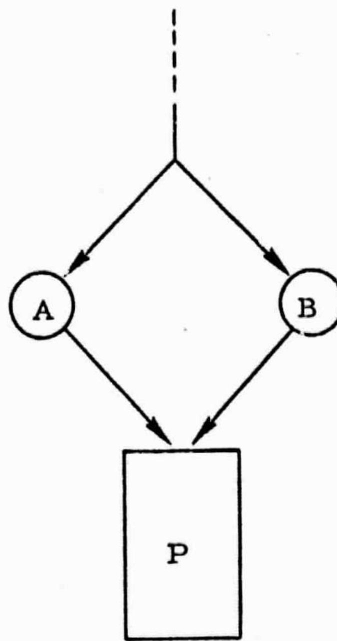


Figure 3-5B

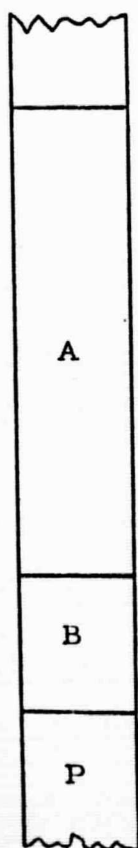


Figure 3-5C

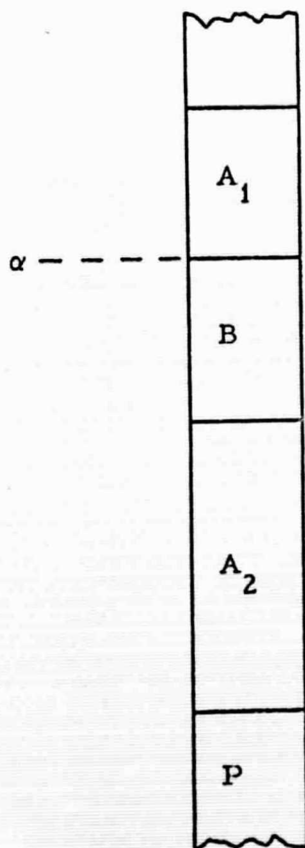


Figure 3-5D

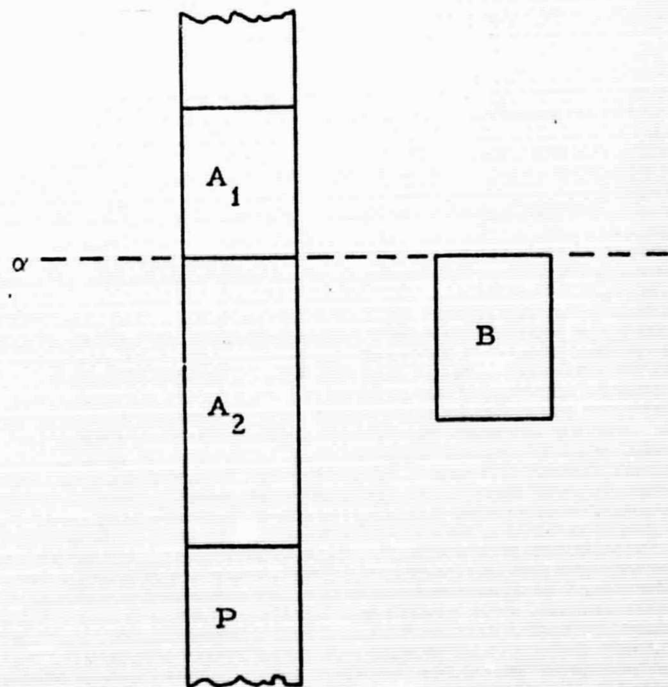


Figure 3-5E

Figure 3-5. Commutativity and Interruptibility Testing

These conditions are identical to the conditions that must be satisfied to allow B to be executed in parallel with A_2 (Figure 3-5E). The interrupt point α may occur anywhere in A, and the instructions comprising A_2 will always be a subset of the instructions comprising A. Hence guaranteeing that B can be executed in parallel with A guarantees that B can be executed in parallel with any $A_2 \subseteq A$, which guarantees that B can interrupt A at any point. The converse statement is also true.

Checking conditions 1 and 2 does not appear to present any practical problem. For condition 3, a practical procedure would be to check $W(A) \cap W(B)$ first. Only in the case where this is non-null would it be necessary to check intersection with $M_1(P) \cup M_2(P)$, which in general would be much larger than W(A) or W(B).

The conditions for commutativity of two routines have the same structure but with simpler defintions for the R's and W's:

$$R'(B) \cap W'(A) = \varphi$$
$$R'(A) \cap W'(B) = \varphi$$
$$W(A) \cap W(B) \cup R'(P) = \varphi$$

where

$$R'(A) = M_1(A) \cup M_2(A)$$
$$W'(A) = M_3(A) \cup M_4(A)$$

If the M's are substituted into the equations above and redundant terms are eliminated, the terms which must be null are:

Interruptibility	Commutativity
$M_1(A)M_2(B)$	$M_1(A)M_3(B)$
$M_1(A)M_3(B)$	$M_1(A)M_4(B)$
$M_1(A)M_4(B)$	$M_2(A)M_2(B)(M_1(P) \cup M_2(P))$
$M_2(A)M_1(B)$	$M_2(A)M_3(B)$
$M_2(A)M_2(B)$	$M_2(A)M_4(B)$
$M_2(A)M_3(B)$	$M_3(A)M_1(B)$
$M_2(A)M_4(B)$	$M_3(A)M_2(B)$
$M_3(A)M_1(B)$	$M_3(A)M_3(B)(M_1(P) \cup M_2(P))$
$M_3(A)M_2(B)$	$M_3(A)M_4(B)(M_1(P) \cup M_2(P))$
$M_3(A)M_3(B)(M_1(P) \cup M_2(P))$	$M_4(A)M_1(B)$
$M_3(A)M_4(B)$	$M_4(A)M_2(B)$

Interruptibility	Commutativity
$M_4(A)M_1(B)$	$M_4(A)M_3(B)(M_1(P) \cup M_2(P))$
$M_4(A)M_2(B)$	$M_4(A)M_4(B)(M_1(P) \cup M_2(P))$
$M_4(A)M_3(B)$	
$M_4(A)M_4(B)$	

The above demonstration indicates that the conditions required to determine whether two program segments can be mutually interruptible are not more numerous or more complex than those necessary to establish that they are commutative.

3.3.2 Memory Locks

The preceding discussion of interrupts and their effect on program verification and validation indicates that a significant problem exists in preventing one task from referencing another task's data and thereby causing unwanted task interaction. In multiprogramming environments, memory locks are commonly used to prevent such undesirable interaction of independent programs. For the Space Shuttle application, memory locks would insure orderly interaction of related tasks. The purpose of a memory lock is to temporarily isolate certain regions of memory from certain tasks.

Four types of memory locks exist: read, write, execute, and total. A read lock prevents a task from reading a region of memory as data and is useful to prevent data from being used while being modified. A write lock prevents a task from writing in a region of memory and is useful when a task is accessing data which must not be disturbed by other tasks. An execute lock prevents a task from executing instructions in a region of memory. Such a lock is useful to prevent two processors from executing a routine simultaneously. Since execution of an instruction requires that memory be read, the read and execute locks may be combined into a single read lock. A total lock prevents any kind of interaction between a task and a region of memory, whether read, write, or execute.

The desirable size of the regions of locking is highly dependent on details of the interacting tasks. The simplest scheme is to have uniformly sized blocks to which the locking applies. If such blocks are too large, much more memory may be locked than is really desired. If too small, locking overhead may be unacceptably high. For the Space Shuttle application, with its relatively fixed set of tasks, it might be feasible through microprogramming to use variably sized locking regions, with the sizes tailored to specific sections of data or specific routines.

To make any routine A secure from undesirable interrupts, the following procedures may be used:

1) Read locks are placed on all locations in the set W(A) at the beginning of A. After the last write statement in A for a given location, the lock on that location is lifted.

2) Write locks are placed on all locations in the set R(A) at the beginning of A. After the last read statement in A for a given location, the lock on that location is lifted.

3) Write locks are placed on all locations in the set W(A) at the beginning of A and are lifted at the end of A.

To see that these locking procedures are effective, recall the three conditions for interruptibility:

1) $R(B) \cap W(A) = \varphi$
2) $R(A) \cap W(B) = \varphi$
3) $R(P) \cap W(A) \cap W(B) = \varphi$

Procedure 1 exactly satisfies condition 1, since the read lock on W(A) halts any routine B only if B tries to read from W(A). Procedure 2 exactly satisfies condition 2, since the write lock on R(A) halts any routine B only if B tries to write into R(A). Procedure 3 oversatisfies condition 3. The write lock on W(A) will halt any routine B if B tries to write into W(A). This is equivalent to the condition $W(A) \cap W(B) = \varphi$, which implies that condition 3 is true. A procedure simpler than all three of the above is the place a total lock on R(A) and W(A) for the entire duration of routine A. The disadvantage is that other routines will sometimes be blocked unnecessarily.

Implementing memory locks through microprogramming would combine the efficiency of hardware with the flexibility of software. Memory locking implemented entirely in hardware would be undesirable because of the redundancy requirements. Implementation entirely through software would be slow and possibly unreliable. For example, if testing and setting of a lock flag takes more than one instruction, it is possible for two tasks to set a lock at the same time. These tasks would then proceed to use the memory as if each one had exclusive access. The result is software failure. The minimal feature is a "Test and Set/Branch" instruction, which in single instruction tests a word and sets it to 1 if it was 0 or branches to another address if it was already 1.

For microprogram implementation, a binary matrix can control access to memory. One dimension represents tasks and the other represents the lockable regions of memory (there may be no need to provide for the locking of all main memory). A 1 in the matrix indicates a task may access the memory region. A 0 indicates the task is locked out.

When a task finds itself locked out, there are two alternatives open: (1) stay on the processor and wait until the lock is removed, or (2) relinquish the processor to a waiting task and go onto a lockout waiting list. This list would be scanned by the executive whenever a lock is removed to see if any of the tasks can continue. The choice between the two procedures depends on how long the lock can be expected to exist, and how long it takes to swap jobs on a processor. It may be desirable to designate both short locks and long locks. For a short lock, the task would follow the first procedure; for a long lock, it would follow the second.

A pointer lock may be useful when a linear data structure is shared by two or more tasks. A region of memory is designated as controlled by pointer lock. Memory locations up to the pointer may be accessed. Locations past the pointer are locked. Figure 3-6 depicts the scheme.

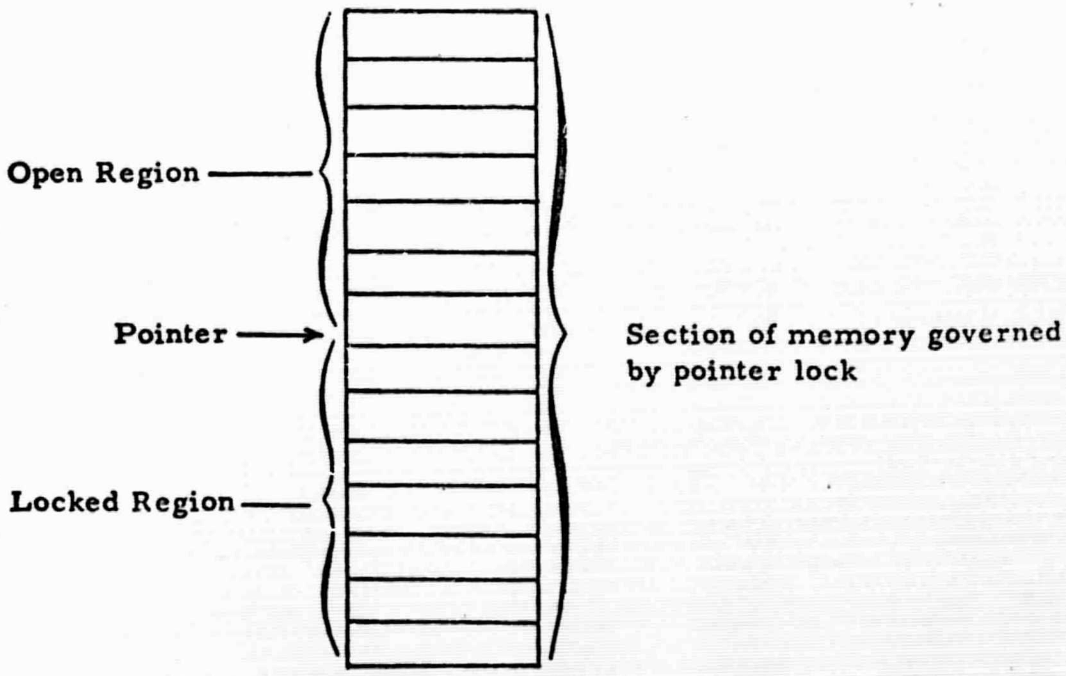


Figure 3-6. Pointer Locks

The task producing the data advances the pointer as it is refreshed (possibly automatically via microprogramming a special "Store and Advance Pointer" instruction). Tasks accessing the data are allowed to proceed up to the pointer but no further. They need not wait until the entire data section is refreshed before beginning to use the data, which would be the case if a block lock were used.

Prevention undesirable task interaction by means of memory locks results in a degradation in efficiency because a task's execution might be inhibited for a period of time until that lock is removed. This might be tolerable for a low-priority task attempting to reference data being computed by a high-priority task, but it would be intolerable in the converse situation. Other architectural features such as indirect addressing and single instructions which accomplish the transfer of an entire block of data also provide a partial solution to the problems of task interaction. However, the employment of these features to maintain multiple copies of data items which must be shared requires additional memory. Whether the memory locks that have been discussed or data duplication techniques are employed depends on both the time delays that can be tolerated and the size of the data arrays being shared.

3.3.3 Register Stacks

Any computer program of the complexity of that to be encountered in the Space Shuttle application involves a hierarchical structure of subprograms, routines, subroutines, statements, and expressions. In such nested structures an operation at one level cannot be completed until those at the next lower level are completed. An examination of the nested structures existing in a typical program suggests some architectural features which facilitate a solution of the problems they present. One such architectural feature is a push-down stack mechanism for registers.

In programming, nested structures are commonly observed in:

1) Arithmetic and logical expressions
   e.g., (A + B) * (C + D) or . NOT. (A. OR . B)

2) Subroutine calls
   e.g., program A calls subroutine B, which calls subroutine C. and so on

3) Interrupt levels
   e.g., program A is interrupted by program B, which is interrupted by program C, and so on.

The existence of one or more hardware-implemented stacks, together with related stack-manipulation instructions, can simplify or eliminate the software chores usually required in the evaluation of arithmetic and logical expressions. It also minimizes the number of instructions needed to save and restore partial results. A hypothetical implementation will illustrate the way in which a stack-oriented computer would work.

Such a system permits a simple one-to-one correspondence between the reverse Polish notation for expressions and their object code. The string AB+CD+* becomes the object sequence:

```
LOAD A
LOAD B
ADD
LOAD C
LOAD D
ADD
MULTIPLY
```

For a machine with stacked registers, techniques exist which make it possible to generate object code directly from the infix notation of a higher level language. Thus the compiler for a stack register machine can be easier to write and will run faster than a compiler for a machine with conventional arrangement of data registers. The object code produced by the compiler will compare more favorably with assembly code than is usually possible in a conventional computer organization. Consequently, use of a higher level language is more attractive for such a machine.

Subroutine transfers and returns can be performed faster and programmed more easily if special subroutine branch and return instructions are provided which use a hardware stack. A return pointer and other necessary information are placed on a stack before transferring to the subroutine, and removed to effect the return.

If one level of indirect addressing is added in conjunction with the stack, all task routines and subroutines can be made automatically reentrant. As a routine or subroutine goes into execution, either by a subroutine call or through initiation by the task scheduler, a marker is placed on the stack and the parameters and data for the routine are entered above it. Addressing of data local to the routine is relative. Prior to data fetch, the address of the marker is added to the relative address to obtain the absolute address. Essentially, the details of reentrant programming have been built into hardware.

Similarly, the saving of the state of the computer when an interrupt occurs is simplified in that only a return address and the value of the pointer need be saved. When an interrupt occurs to resume execution it is only necessary to reset the pointer to its previous value and return to the address saved. Interrupts are thus handled in a fashion equivalent to subroutines.