1

Design of an Engineering Flight Simulator

I have my hopes, and very distinct ones, too, of one day getting cerebral phenomena such that I can put them into mathematical equations.

Ada Lovelace

COPYRIGHTED MATERIAL

1.1 The Evolution of Flight Simulation

The Link Trainer is generally regarded as the forerunner of flight simulation. Ed Link had worked in his father's factory in Binghamton, where they manufactured air-driven pianos and church organs. Having gained his pilot's licence in the late 1920s, Link applied his knowledge of pneumatics to the construction of a flight trainer (Link, 1930), using compressed air to tilt and swivel the cockpit and to drive pressure gauges to replicate aircraft instruments. His invention was remarkable in several ways:

- It was the first time pilots could undertake instrument training in a synthetic device rather than an aeroplane.
- The flight trainer was based on pneumatics.
- The aircraft motion was based on an empirical model rather than a mathematical model.

Six Link Trainers were purchased by the US Army Flying Corps in the early 1930s, following several fatalities attributed to a lack of skill in instrument flying, establishing the benefits of a synthetic training device. The case for simulation was further reinforced during the Second World War with many allied pilots trained in instrument flying on the 'Blue Box', as the Link Trainer was affectionately known.

The limitation of the Link Trainer was that its model of aerodynamics and flight dynamics was based on a simple approximation to aircraft performance. It was the development of the operational amplifier in the 1940s, using thermionic valves, that enabled the differential equations in aircraft dynamics to be modelled. Analogue computers, constructed from operational amplifiers connected via patch boards, enabled complex sets of differential equations to be solved in many branches of engineering (Korn and Korn, 1965), although these computers required daily calibration and considerable care was needed to scale the equations to operate within the voltage range of the equipment.

It was not until the 1970s that the speed of digital computers was sufficient to solve the differential equations in flight simulation 50 or 60 times per second. This iteration rate, often known as the frame

Flight Simulation Software: Design, Development and Testing, First Edition. David Allerton.
© 2023 John Wiley & Sons Ltd. Published 2023 by John Wiley & Sons Ltd.
Companion Website: www.wiley.com/go/flightsimulationsoftware

rate, underpins all flight simulation and consequently, flight simulation demands high-performance computers. During the 1970s and 1980s, this performance was met by the minicomputers of the period.

Since that time, the performance has been increased as a result of developments in computer architecture and microelectronics (Moore, 1965). Firstly, processors the size of a postage stamp are capable of executing hundreds of millions of instructions per second, while at the same time the cost of processors has reduced dramatically. Secondly, with the availability of computer networks, the overall processing speed required in simulation can be achieved by connecting computers as a distributed architecture using a local network. Thirdly, the graphics needed for aircraft displays and image generation is now available from off-the-shelf graphics cards with multiple cores.

It is primarily these advances in processing speed that have enabled airlines and military organisations to provide very realistic pilot training (Allerton, 2000), albeit with simulators costing over $10 million, but where the hourly training costs are often less than one-tenth of the cost of airborne training. These advances in computer technology and flight simulation have been remarkable (Allen, 1993). Flight simulators are used by all major airlines, and regulations have been approved for worldwide training using flight simulators. Similarly, the training of military pilots in flight simulators has increased while reducing both the cost of training and impact on the environment.

Concomitant with these developments in flight training, simulation has taken on a pivotal role in engineering design and development. In industry, control systems are designed with the aid of analysis tools to determine the stability and response of complex systems. In electronics, circuit simulation tools enable circuits to be evaluated prior to the relatively expensive process of manufacturing integrated circuits. In mechanical engineering and aerospace, computer-aided design packages enable designs to be captured on computer screens, visualising and animating designs to facilitate the rapid development of concepts.

These developments in engineering have coincided with changes in the aerospace industry. Aircraft manufacturers have moved away from the traditional departments of aerodynamics, structures, propulsion and avionics towards much more integrated teams with a systems approach to design. Aircraft are viewed as platforms of sensors and computers to enable an aircraft and its flight crew to complete a mission in terms of efficiency and reliability. For many aerospace companies, synthetic environments, including flight simulation, are nowadays a major component in the design of aircraft, covering proof-of-concept and feasibility studies and enabling comparative studies to be undertaken (Allerton, 1996). In some organisations, full mission analysis is undertaken in synthetic environments and the flight simulator is just one of a set of synthetic tools to develop and analyse complex scenarios, which would be impractical with live aircraft.

For manufacturers of civil aircraft, an *iron bird rig* (Jacazio and Balossini, 2005) is used in the development and testing of aircraft systems and actuators. The aircraft systems are set out in a large building with actuators mounted in test rigs. Although the flight deck is a synthetic component, actual aircraft equipment is connected to the simulator, including cables, connectors, pipes, power supplies, avionics equipment, databuses and actuators. During simulated flight, the actuators respond as they would in the aircraft and each actuator can be monitored or loaded or failed in order to test the response of the aircraft systems. The iron bird rig is the final stage before flight testing and is largely used to prove that the software models developed in a laboratory meet the aircraft requirements for performance, stability and reliability.

In this modern role of simulation in the design, development and testing of aircraft and aircraft systems, the simulator is often referred to as an engineering flight simulator rather than a flight simulator training device (FSTD). Its use is not to train flight crews but to provide a tool to improve the design of systems and to validate these designs thoroughly prior to manufacturing. Of course, in tests involving a pilot, the engineering flight simulator has many of the characteristics of an FSTD.

In summary, as aircraft systems have increased in size and complexity, the dependence on simulation as an essential tool in developing and testing prototype systems has also increased. The alternative method, of designing an aircraft and flight testing a prototype, with the possibility of faults only becoming evident once the aircraft is in service (Cohen, 1955), is no longer seen as a viable option. Simulation is very much the focal point of modern system design and, consequently, the quality and accuracy of the simulation software will have a major impact on the success of the design. In further sections, the software used in simulation will be explored in more detail, but for now it is fair to assume that simulation is here to stay (Allerton, 2010) and all system designers need to appreciate both the capabilities and pitfalls of using flight simulation in aircraft design and development.

1.2 Structure of a Flight Simulator

The structure of the majority of flight simulators is shown in Figure 1.1. Although this is a hardware diagram of modules and interconnections, it can also be viewed as a model of the software modules and interfaces. The main modules are shown as rectangles and the databases are shown as ellipses. Note the direction of the arrows, implying that some modules generate data for the equations of motion, whereas others use data produced by the equations of motion. The most significant point of this diagram is that the equations of motion module is the focal point of the simulator and is connected to all the other modules.

Although the use of flight simulators in civil and military training differs considerably from flight simulators used in engineering research, the core software is common to most flight simulators. The major variation is the number of visual channels and methods of projection. Invariably, the motion platform is omitted in an engineering simulator and is usually replaced with a G-cueing seat in military simulators (White, 1989).

The modules are shown as individual systems, but many comprise sub-systems specific to the module function. Note also that the term *database* is not used in the common usage used in computing; rather, they are databases containing data that is specific to a module and, in this sense, they are application-specific files loaded when the simulation starts.

From a modelling perspective, the equations of motion module is the core of the simulator. It contains the state of the simulation, updates the aircraft dynamics at the frame rate of the simulator and acquires the inputs to compute the forces and moments applied to the vehicle. In turn, the forces and moments are used to compute the accelerations, velocities and positions of the vehicle. Often these equations are referred to as six-degree-of-freedom (or 6-DOF) equations because they compute the linear state of the vehicle in three axes and the rotary state of the vehicle in three axes. The equations of motion module is also responsible for transforming forces, accelerations, velocities and position between axes, to provide the simulator state data in an appropriate form for the various modules. Generally, the equations of motion are applicable to any form of aircraft, where the aircraft-specific information is retained in the aerodynamic model, the gear model and the engine model.

The aerodynamic model contains information to compute the aerodynamic forces and moments of the airframe, which is unique to a particular aircraft. For many aircraft, the database is provided by the manufacturer and contains the aerodynamic data for the aircraft for the complete flight envelope in all configurations. In addition to the aerodynamic data, it includes test data used to validate the simulation. The quality of the flight model is dependent on the quality of the flight data, and during qualification of a simulator, the performance and handling of the simulated aircraft will be compared with the validation data provided for the simulator.

The gear model is a mechanical model of the undercarriage assembly of the aircraft, in particular the springs, oleos (dampers), brakes and tyres. During the take-off and landing roll and taxiing, the

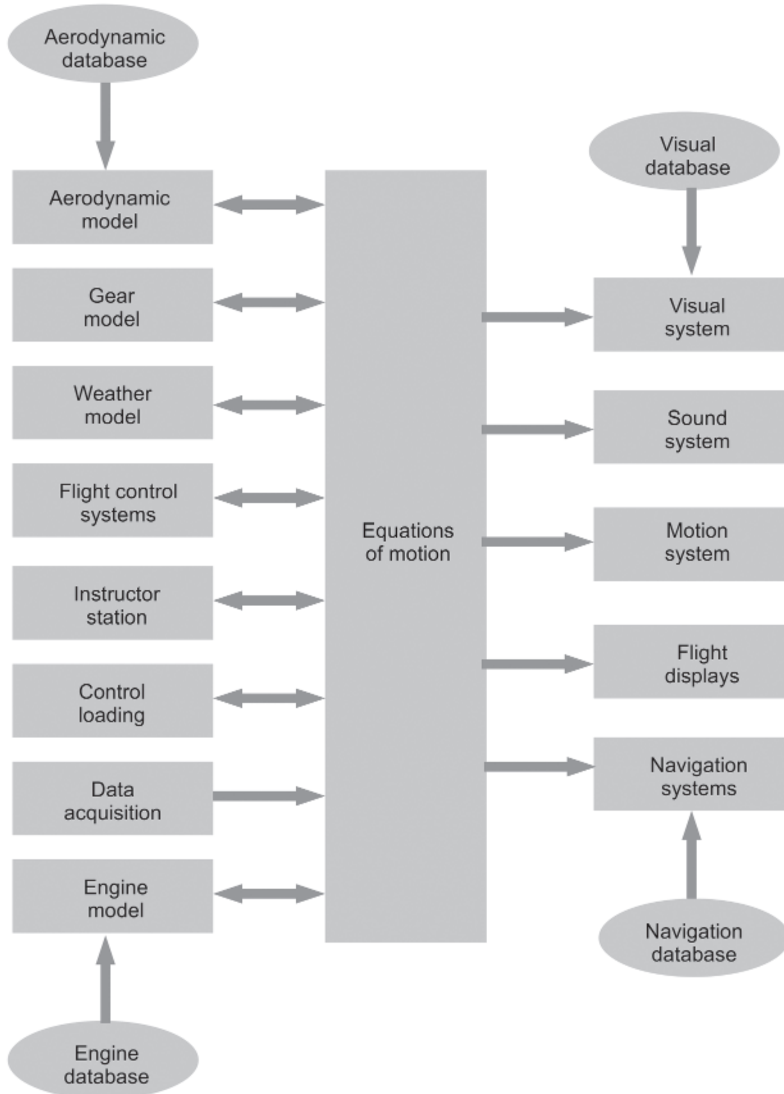


Figure 1.1 Structure of a Flight Simulator.

interaction between the rudder pedals, toe brakes and tiller results in forces and moments in the undercarriage assemblies, which are transformed to the aircraft frame. The models will include tyre scrubbing and scuffing, brake fading and overheat and possibly tyre burst. As the take-off and landing are critical phases of flight, the gear model is an important component of simulation and extensive tests of take-offs and landings are undertaken during qualification of a flight simulator.

As temperature, pressure and density of air in the troposphere vary with altitude, these parameters must be modelled correctly. Air temperature affects engine performance, aerodynamic performance varies with air density and air pressure is used in the air data computer computations and flight instruments. In addition, a weather model provides winds, from either meteorological data or a generic model, turbulence, microburst data, precipitation (used in modelling weather radar) and wind shear.

A modern civil transport aircraft has a flight management system (FMS) and flight control systems (FCS) to provide automatic modes to aid navigation. In a flight simulator, these systems are simulated to enable flight crews to operate the equipment in exactly the same way as the aircraft. The FMS commands the FCS, which drives the primary aircraft flight controls to manage airspeed, altitude, rate of descent and heading. The response of the FCS must match the aircraft systems very closely and, in some cases, an avionics database is also provided for the FMS and FCS.

A flight simulator for a civil transport aircraft used for pilot training will be managed by an instructor who sets flight conditions, introduces failures, monitors pilot performance and provides debriefing for flight crews. In a civil simulator, the instructor is positioned towards the back of the flight deck at the instructor operating station (IOS) and is provided with a screen to monitor the aircraft track, enter settings and introduce environmental conditions, typically via menu selection. The role of the simulator instructor is very important – they monitor the flight crew operations closely, to intervene as part of instruction but not to interfere with operations in an unnatural way. The design of the IOS and its capability can influence the quality of instruction considerably. Although there is no official standard for instructor station design, most modern instructor stations are very similar in terms of their capabilities and user interface.

In an aircraft, the aerodynamic loads on control surfaces vary with airspeed and acceleration and this dynamic loading of the primary controls is achieved in flight simulators by means of hydraulic or electrical actuation. In addition, the throttles and trim wheel may also be driven by an actuator, typically a servo-motor. The response of the control loading system, which is a dedicated system on most flight simulators, is much faster than the main frame rate. A major concern with active controls is the safety of the flight crew being physically close to active controls and safety interlocks are provided in both hardware and software to ensure that the movement of the controls cannot cause harm to a pilot as a result of inadvertent activation.

For the engine model, the manufacturer will provide detailed data of engine performance and engine dynamics. A modern turbofan engine includes a full authority digital engine control (FADEC) system and details of its operation are also provided in the data package. In addition to the provision of data to compute thrust, RPM, fuel flow and engine temperatures, the starting and shutdown procedures (both on the ground and in flight) must be modelled correctly, in order to introduce engine-start problems and to ensure flight crews have a full understanding of engine operation to cope with partial and full engine failures. The engine model is a combination of thermodynamics (gas flows) and the dynamics of turbomachinery rotating at very high speeds. Much of this data is proprietary and licensed to the airline operating the simulator.

A flight simulator may have several hundred wires connecting the levers, knobs, selectors, switches and controls. These connections are a combination of analogue and digital inputs which must be acquired every frame. Often dedicated input/output (I/O) hardware is provided to capture input signals at high data rates and with sufficient resolution. The actual sampling and conversion may take several microseconds per signal and consequently, special-purpose hardware is used for data acquisition with high bandwidth transfers to the memory of the simulator computers, in order to minimise delays associated with data acquisition.

In a modern flight simulator, three or four independent image generators (IGs) render the external scene viewed from the flight deck or cockpit. The actual scenery is stored in a visual database and loaded by each IG at run-time. The scene is typically formed from millions of coloured and textured triangles and the IG performs 3D graphics operations to display these images at 50 frames per second (fps) (or higher) with resolutions approaching 2000 × 2000 pixels per channel. The images are projected, merged and blended to form a continuous wrap-around display of scenery which includes dynamic entities. The conditions displayed by the IGs should match the weather

conditions, varying from thick fog to perfect visibility, and produce lighting according to the (simulated) time of day. A considerable amount of development goes into the production of detailed databases for airports and terrain, and real-time 3D graphics algorithms ensure highly detailed images are rendered at the visual system frame rate.

In addition to providing visual cues, sounds that are audible in flight need to be replicated in a flight simulator. Nowadays, the accepted method of sound generation is to carefully record sounds on the flight deck and then play back these sounds using a sound card which generates a sound signal from data stored in a buffer in a recognised format, for example, a *wav* file. Fortunately, most of the requirements for sound generation are met by the capabilities of modern sound cards. Typically, a database of sounds is accessed to generate sounds appropriate to the flight conditions.

Many of the accelerations on the human body occurring in flight are very different from everyday movement and motion cues in a simulator are matched to visual cues to emphasise the sense of motion in a simulator. This is achieved in simulation by attaching the flight deck to six hydraulic (or, more recently, electrical) linear jacks, which move independently in order to replicate the three linear movements and the three angular rotations. The main problem is that the length of movement of the jacks is of the order of 2 m and there is an inevitable difference between motion perceived in an aircraft and motion in the simulator, which cannot be sustained. Nevertheless, for pilot training, a motion platform is a requirement for the qualification of full flight simulators. For engineering flight simulators, the motion platform is usually omitted on the basis that it contributes little to the design and testing of aircraft systems.

Modern military and transport aircraft have flat screen displays, known as electronic flight instrument systems (EFIS) rather than the mechanical instruments used in smaller aircraft. In flight simulation, it is possible to use 2D computer graphics to emulate EFIS displays. The difficulty with emulation is to ensure that the frame rate is maintained at all times. An option is to use actual aircraft displays and generate the appropriate inputs but, invariably, this is a far more expensive solution.

Finally, navigation is an essential part of aircraft operations and, in simulation, the aircraft may fly several thousand miles using en-route navigation and guidance from radio transmitters, satellites and inertial sensors. The operation of the navigation equipment and alignment of worldwide radio aids with aircraft routing must be accurate to the tolerances of airborne navigation. The navigation database contains all the beacon locations, frequencies and ranges, runway layouts and information that is loaded into an FMS, including airways and departure and arrival information (SIDs and STARs). In addition, failure modes must also be simulated correctly.

This brief description of a typical flight simulator outlines the modules and their functions. There is no unique organisation of a modern flight simulator. Nevertheless, these modules are common to most flight simulators and the software used in these modules is designed to meet the requirements of each module. Once the overall structure is determined, consideration can be given to the design of the software structures, in particular to establish the inputs required by software modules and their connection to the simulator computers.

1.3 Real-time Flight Simulation

1.3.1 The Concept of Real-time Computing

Most programs and applications running on computers are captured as source code written in a high-level language, compiled to machine instructions, linked with relevant libraries and then loaded into memory and executed. For most users, the emphasis is on correctness rather than

performance. For example, in computational fluid dynamics, the computation of flow fields may take several hours and the speed of execution is mainly influenced by the efficiency of the compiler and the speed of the processor (or cores). In other applications, particularly where there is human interaction with the software, performance is a major consideration.

In the 1980s, several vendors produced flight simulator games for home computers. Many experienced pilots found these programs to be much harder to fly than airline simulators or the actual aircraft. The controls were flimsy and the PC screens were small but the main criticism was the speed of the PC graphics hardware to update the displays, particularly the external view. The games developers traded off visual fidelity (realism of the displays) for the frame rate to the extent that many games updated at only 3–5 fps, causing pilots to over-control their inputs. From an engineering perspective, this was a well-known problem in sampled-data theory.

There are many examples of systems that must meet tight timing constraints, which are referred to as *real-time* systems. A system is a real-time system if the inputs are captured and responded to within a defined time. For example, a computer in a bottling factory may be required to detect the position of a bottle on a conveyor belt, select a cap and actuate a mechanism to secure the cap to the bottle within one-tenth of a second. This repetitive process must be guaranteed to apply a cap within 100 ms, at all times and under all conditions, and the software will be validated to ensure that the real-time requirement is fully met.

The situation just outlined is also common in flight simulation. The positions of the flight controls and the various levers, knobs and switches are acquired, the equations of motion are computed for the airframe and the engines, and the results are output as computer graphics on displays and possibly to actuators to position the motion platform. A typical frame rate in real-time simulation is 50 Hz, giving a frame time of 20 ms. By completing the computations within this frame time, the perceived motion is smooth with no apparent discontinuities, the simulation of the dynamics is computed to an acceptable accuracy and the sampling rate is matched to the time constants of the flight dynamics.

There is one significant difference from the industrial example – the pilot interacts with the software. In a flight simulator, the pilot acquires information from the simulator displays and responds to move the inceptors. The delay resulting from acquiring visual information, cognitive processing and actuating muscles in the hands, arms and legs is of the order of 0.2 s (McRuer, 1995). At 50 Hz, a pilot will probably not detect any discontinuities or irregularities in the perceived motion of a transport aircraft but, with the simulation of agile fighter aircraft, the time constants of the dynamics may be much smaller, necessitating a faster frame rate. Similarly, hydraulic actuation of a motion platform may necessitate an update rate in excess of 1000 Hz to ensure smooth (and indiscernible) movement of the platform.

In real-time software, the time to respond to an input event must be guaranteed at all times (Burns and Wellings, 2001). However, synchronisation and timing of code in high-level languages are mostly provided by operating system functions. This distinction is important because responsibility for the real-time response is delegated to the operating system rather than user software, ensuring that:

- the operating system is responsible for the interface with hardware devices, providing optimal performance of data transfers;
- transfers and errors are managed in a clear and consistent manner;
- the operating system decides the order and priority of accesses to external devices.

The major constraint in a real-time system is that there is a finite number of instructions that can be executed during one frame. Consequently, in real-time systems, emphasis is focused on the

speed of executing software, and advances in processor architecture have benefited flight simulation considerably:

- Processor cycle times have reduced from several microseconds to tens of nanoseconds over the last 20 years or so.
- Modern processors contain multiple cores operating in parallel. In applications where parallel streams can be identified and allocated to specific cores, considerable gains in processing speed are achievable.
- Instruction caches allow pending instructions to be fetched while the processor is executing the current instruction; fetching an instruction from the cache is much faster than fetching instructions from memory (Wilkinson, 1996). However, when functions are called or there is a jump in the execution path of the code, the instruction caching restarts.
- Similarly, data caches are used to pre-fetch data from memory so that frequently accessed data is available in the cache rather than via memory. When the region of accessed data changes, the data caching restarts.
- Direct memory access (DMA) allows data to be transferred between memory and external devices while the processor is executing instructions. The processor is responsible for initiating the transfer and responding to the completion of the transfer but the processor executes instructions in parallel with DMA transfers.
- Modern compilers generate code that is optimised for specific instruction sets and register sets of a processor, significantly increasing processing speeds as variables are mostly accessed via machine registers rather than memory.

However, these advances do not ensure the real-time requirement to complete operations within a frame; they enable more computing to be completed during the frame. The performance of the operating system in a real-time system is as important as the performance of the user code.

1.3.2 Operating Systems

A main role of an operating system is to facilitate data transfers between applications and external devices. The user interface is simplified because the user software needs no detailed knowledge of the hardware, the software interface is well-defined and much of the software interface is common to all devices. Typically, transfers comprise five stages:

1) Establishing linkage to the hardware.
2) Setting the device to perform specific transfers, for example, the direction of the transfer and the number of bytes of data to transfer.
3) Initiating a transfer.
4) Completing the transfer, particularly to establish if the transfer succeeded.
5) Closing the linkage to the hardware.

Stages 2–4 enable transfers to be repeated while the device is, in effect, connected to the user software. Between stages 3 and 4, while a transfer is under way, the processor executes other instructions rather than simply waiting for the completion of the transfer. The occurrence of stage 4 is normally an interrupt, where the processor stops its current process in order to respond to the interrupt, checking the status of the transfer and possibly initiating another transfer.

This sequence of events is common to all operating systems. For example, in a desktop computer, file transfers to the hard drive are performed by establishing a channel between an area of user memory and a region of the hard drive, defining the size of the data to be transferred and the

direction of the transfer and activating the transfer. On completion of the transfer, the operating system checks that no errors occurred and can signal to the user process that the transfer is completed and the user code can proceed.

Depending on the operating system, it may be managing hundreds of processes. However, only one process can be active at any time. Many processes may be dormant, waiting for some event to occur or being unable to proceed until an I/O transfer is completed. Within the operating system, the *scheduler* is responsible for switching between processes and the characteristics of the scheduler define the behaviour or response of the operating system. The scheduler is able to start and stop processes, saving critical information, so that a process can subsequently resume. Of course, this saving and restoring of the process state is an overhead. Within a frame, each rescheduling of processes may take several hundred instructions which, in terms of a user process, is lost time.

The other responsibility of the scheduler is to select the most appropriate process to run and there are many variations of operating systems. For example, in a time-shared system, the scheduler keeps track of the amount of time spent by each process and tries to give each process a fair share of the overall processing. In most operating systems, I/O processes are given a higher priority than user processes to maximise the throughput of data transfers. However, in real-time applications, it is likely that some inputs are more important than others and should be responded to more quickly. A strictly real-time system allows the user to assign the response to inputs in terms of priority, where the interrupts from specific devices are responded to, not in chronological order, but in order of predefined priority.

There is an alternative methodology to achieving a real-time response which is used in most flight simulation software. In this case, a fixed frame rate is defined and all inputs, all processing and all outputs are completed within this frame time, for every frame. For example, the inceptor inputs for the elevator, ailerons and rudder are acquired, the flight dynamics are computed as a result of the new inputs and the displays and visual system are updated. Similarly, the engine controls are sampled, the engine dynamics are computed and the outputs are used to update the engine displays. It is essential that all the software must be completed within the defined frame time; it is not acceptable for any software to exceed this limit under any conditions. Any other processes which consume processor time must be minimised to avoid overrunning the frame time limit.

1.3.3 Latency

In flight simulation, the pilot control inputs are sampled, the system dynamics are updated and the outputs are visualised. If the delays between a pilot input and the updating of the displays and the visual system exceed a threshold, the response of the simulator can differ appreciably from the response of the aircraft. For most flight simulators, the maximum frame time is 20 ms, giving a minimum frame rate of 50 fps. The challenge for the simulator developer is to organise the software to ensure that the simulation updates at 50 fps, under all conditions.

The inputs may be movements of a mouse and the outputs may be to a graphics display. Alternatively, inputs can be sampled from analogue or digital devices where the output drives an actuator. In both cases, the input, computation and output must occur within a single frame. From the user's perspective, there is a finite delay between entering an input and the result being applied to a screen or an actuator, which is termed *latency*. For applications involving human input and displays, the latency should not exceed 60 ms, otherwise discontinuities or lags in a response are discernible by a human operator and will affect the response of the operator. Although a frame time of 20 ms is well below this threshold, the actual latency may be increased by sampling stale

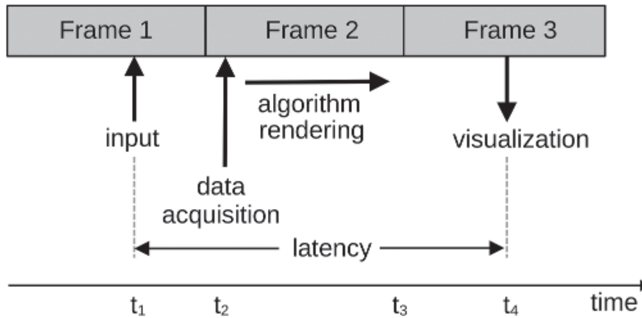


Figure 1.2 Latency.

data (e.g. an input from a previous frame) or the mechanical bandwidth of an actuator or the rendering of a display, as shown in Figure 1.2 for a real-time graphics application.

The input is applied in frame 1 at t_1, but it is not sampled until the start of frame 2 at t_2, the response is computed and the graphics are rendered during frame 2, meeting the 20 ms frame time. However, with real-time graphics, the graphics output is not written to the display (or projector) until frame 3. The acquisition of the image occurs during frame 3 (possibly averaged to the mid-frame time) at t_4. In this example, the latency is not 20 ms, but is closer to 40 ms. There are, in fact, two latencies, the delay between sampling the input and the computed output (t_3–t_2), where the state computed in frame 2 will be used in frame 3 and the delay between the actual pilot input and the cognitive acquisition of the display by the pilot (t_4–t_1). Although increasing the frame rate can reduce the latency, it implies an increase in processor speed and faster data acquisition hardware but cannot eliminate the latency. Considerable care is needed in simulation (and aircraft FCS) to minimise latencies, particularly where they include delays (e.g. additional frames in rendering graphics), which are independent of the user software. In the case of image generation systems, it is very important to clarify the number of frames between rendering the image and its subsequent projection, particularly as some graphics cards require additional frames for post-processing.

1.4 Distributed Computing

In the past, with high unit costs for a computer, emphasis was given to centralised computing with minimal external connections. When external data was connected to a computer, it was often in the form of serial or parallel interfaces. This situation changed with the arrival of the microprocessors used in desktop devices and the subsequent availability of interconnection to a local network, particularly Ethernet, which has become ubiquitous with relatively cheap cards providing data rates in excess of 100 million bits per second (Mbps). Although this data rate is impressive, it is important to bear in mind that if large numbers of users are connected to the network and are making heavy use of it the bandwidth of the network is, in effect, shared between the users, and the data rate between any two users on the network is reduced owing to the loading of the network.

Ethernet has, arguably, revolutionised computing, offering many advantages:

- An Ethernet connection is cheap and simple.
- Large packets of data can be transmitted at relatively high data rates.
- Communication across a network is very reliable.

- Ethernet cables are lightweight and the connectors are small.
- Wireless forms of Ethernet are available for transmission around buildings.
- The signal uses Manchester bi-phase encoding, enabling both the data and the clock to be recovered from two wires.
- The encoding method enables spikes, burst noise and 'stuck-at-one' faults to be detected.
- The packets include a 32-bit cyclic redundancy checksum (CRC); the likelihood that a data error will *not* be detected is of the order of 1 in 2^{32} (1 in 4×10^9) transfers with computation of the CRC implemented in hardware.

Distributed systems offer major advantages in comparison with centralised systems, particularly for applications in simulation:

- If an application is partitioned across n computers, there is a potential speed improvement by a factor of n, although, in practice, the overhead of managing the interconnections reduces this factor.
- Distributed systems are scalable – extra computers can be added to the network to increase processing power or networks can be connected to other networks to increase interconnectivity.
- Conventional software validation can still be applied to individual processors.
- A distributed system can be more fault-tolerant – failures within one computer can be accommodated, possibly by dynamic reconfiguration of the network.
- Standards are published for software interfaces for networks, assuring a high degree of software portability and interoperability if the network hardware is changed.

Nevertheless, distributed systems can also introduce problems:

- Failures of the network can stop all transmissions.
- There is a finite delay in sending data from one computer to another with the potential to introduce latency into computations.
- For the developer, applications have to be redesigned and partitioned in order to operate concurrently, which is not always straightforward with systems with a high sequential content.
- The benefits of parallelism are achieved at the module level rather than the code level.

From the perspective of software design, the major problem is latency or delay in the transmission of data from one computer to another. This delay includes the time to construct the data packet, waiting for access to the network, transmitting the packet on the network and a final delay while the packet is received and copied to the user application. Although these transmissions are likely to be managed by the operating system, in many applications, considerable effort is needed to avoid data becoming 'stale', that is to say ensuring that data used in an algorithm is accessed within some datum of time. This condition is particularly important in real-time systems where the transfers must be *deterministic*, that is to say all transfers are guaranteed to be completed within a finite time, independent of the load on the network or the load on individual computers. However, network latency is difficult to measure; it is the difference between the time measured at the transmitting computer when the message is transmitted and the time measured when the message is received. In practice, the computers on a network are likely to have a local clock to measure time, but there is no guarantee that all these clocks are aligned or synchronised. Consider a series of transmissions for a network of 10 nodes, where each node has to transmit a packet to the other nodes, as shown in Figure 1.3

Each node transmits 9 packets giving a total of 90 packets, although, in practice, not every node may need to send data to every other node. The constraint is that a node only has access to the

Figure 1.3 The Topology of a Local Network.

network when it is not being used by the other nodes for their transmissions. Ethernet offers a major advantage in this situation, that is, packets may be broadcast over the network, known as multicast transfers. A node waits for the network to be available and then broadcasts its packet. Nodes which require the data can read the incoming packet, while other nodes can ignore or discard the packet. In this example, with multicast transfers, the number of transmissions is reduced from 90 to 10.

However, there are three problems with this simplistic explanation of network transfers:

1) How does the transmitter know the network is available?
2) How does the transmitter know that the message was received?
3) How can the transmitter and receiver detect any errors in the transmission of data over the network?

Assuming messages are transmitted asynchronously, any two nodes may compete for bus access. It is possible that the two nodes may detect that the bus is free at exactly the same time and transmit their messages, corrupting the data sent over the bus, which is known as a *collision*. There are two options to avoid collisions; either the transfers are scheduled to avoid contention for the bus or, alternatively, collisions are detected and the transfers are retried later. With high bus loading, the probability of collisions increases and if the transfers are to be deterministic, it is essential to guarantee the transmissions within a finite time.

Ethernet uses CSMA/CD (carrier sense multiple access with collision detection) – if a node detects a collision during transmission, it waits for a random delay and then tries to retransmit its message. The concern with Ethernet is that, although highly improbable, this process could repeat indefinitely, with two nodes blocking the network as they compete for access. In other words, simply transmitting Ethernet packets will lead to collisions and deterministic transfers cannot be guaranteed.

Three protocols are commonly used in networks to ensure deterministic transfers:

1) *Master-slave*: one node on the network is chosen as the master node. Initially, all the other nodes are passive, reading packets. The master node transmits a packet to a slave node, in effect, giving it ownership of the network. The slave node can then transmit its packets, confident that no other node will attempt to transmit a packet. On completion of its transfers, the slave node transmits a packet to the master node, relinquishing control of the network. The master node has a schedule of transfers and repeats this process for all the slave nodes needing to transmit packets. The method assumes that slave nodes are listening for packets from the master node to avoid delays in changes of bus ownership. The packets transferring ownership of the bus are an overhead.
2) *Token passing*: all nodes have a copy of the schedule of transfers. In a cycle of transfers, the first node transmits its packets. On completion of its transfers, it transmits a token (typically a packet) to the second node in the schedule which transmits its packets, passing on the token to

a third node and so on. Only the node with the token is allowed to transmit. Again, the method assumes that nodes take their token immediately. A variant of the method is that the first packet acts as both a token and a data packet. Care is needed to avoid the situation where nodes towards the end of the chain of token passing experience unacceptable delays.

3) *Time division multiple access*: the cycle of transfers is partitioned into discrete time intervals. Each node is given its time slot and is allowed to transmit packets in this period. At all other times, the node is passive. There are three problems with this method. Firstly, no node must exceed its time slot, otherwise two nodes may transmit at the same time. Secondly, each node needs to monitor time very accurately. In practice, the clocks of each node will drift slowly with respect to the other nodes, so that after a long duration, two clocks may differ by a significant amount. Synchronisation of clocks in distributed system is possible but requires a significant number of extra packet transfers to align local clocks. Thirdly, a node may not need all of its allocated slot, in effect, introducing network delays.

Two questions remain. How does a transmitting node know that its packet was received and what happens if the receiving node detects an error? The problem is resolved in the TCP/IP protocol (Donahoo and Calvert, 2001), which is used throughout the Internet. The protocol is based on the receiver transmitting a reply packet to indicate the success of the transfer together with a timeout to detect a null reply. A certain number of retries are permitted in the case of negative replies or timeouts, before the transfer is deemed to have failed. Clearly, the method works, otherwise the Internet would not be viable. The problem for simulation is that, by its very nature, TCP/IP is non-deterministic. The time to complete a transfer is not guaranteed and the number of retries and the message latency is not known a priori.

An alternative protocol often used in simulation is the User Datagram Protocol (UDP), which is a simpler form of network transfer. With UDP, there is no acknowledgement by the receiver and the sender therefore assumes that all packets are received and that there is no corruption of the data in the packet transmissions. At the receiver, incoming packets are checked for consistency, particularly the packet size and checksum. A packet is read if no errors are detected, otherwise the packet is discarded. UDP protocols are only practical in networks where the error rate is low. There is one further consideration with UDP transfers: the temporal order of packets cannot be guaranteed and packets may not necessarily arrive in the order in which they were transmitted, particularly passing through Ethernet switches. In practice, this event is unlikely to occur in dedicated networks where the bus loading is low. If errors can be ignored, UDP has a minimal overhead and can be used in networks where determinism is essential.

The network protocols are set out and defined in the Open Systems Interconnection (OSI) Reference Model (Anon, 1994). This standardisation ensures that user software is independent of the platform, operating system or network card and there is consistency in the software libraries provided to read and write packets on a network. Taking UDP transfers as an example, the user is required to provide a socket to connect to the network. Once set up, the user provides details of the packet to be read or written. The actual management of the transfer is undertaken by the operating system. To transit a packet, the user software provides:

- the IP address of the destination;
- the port number to be used for the transfers;
- the memory address of the buffer containing the packet to be sent.

The actions are summarised in the following code. A data structure **`tx_addr`** of type **`sockaddr_in`** is initialised as follows:

```
memset(&tx_addr, 0, sizeof(tx_addr));
tx_addr.sin_family      = AF_INET;
tx_addr.sin_addr.s_addr = inet_addr(IP_addr);
tx_addr.sin_port        = htons(port);
```

where **IP_addr** is a 32-bit integer holding the destination IP address and **port** is a 32-bit integer holding the port number. The socket is opened by

```
sock = socket(PF_NET, SOCK_DGRAM, IPPROTO_UDP);
```

where **sock** is an integer and a negative result is returned if the socket cannot be opened. The packet is transmitted by:

```
t = sendto(sock, buff, n, 0, (struct sockaddr * &tx_addr, sizeof(tx_addr));
```

where **buff** is an array containing the data to be transferred, **n** is the number of bytes to transfer and **t** is the number of bytes transferred. In the case of an error, **t** is negative. Although this description is simplified, it illustrates the simplicity of Ethernet UDP transfers. A similar function is used to read a UDP packet:

```
r = recvfrom(sock, buff, n 0, (struct sockaddr * &rx_addr, &addr_len));
```

where **r** is the number of bytes received, **sock** is the socket used for reading packets, **buff** is the area of memory where the data is written, **n** is the size of **buff, rx_addr** holds the socket information and **addr_len** is a pointer to an integer containing the size of **rx_addr**.

As an example, consider a simulation using five computers, numbered 1–5. The sequence of transfers in any one frame is shown in Table 1.1, where each row shows the progression of time.

Table 1.1 Sequence of Packet Transfers.

Time	Node 1	Node 2	Node 3	Node 4	Node 5
↓	send pkt1				
		read pkt1	read pkt1	read pkt1	read pkt1
		send pkt2			
	read pkt2		read pkt2	read pkt2	read pkt2
			send pkt3		
	read pkt3	read pkt3		read pkt3	read pkt3
				send pkt4	
	read pkt4	read pkt4	read pkt4		read pkt4
					send pkt5
	read pkt5	read pkt5	read pkt5	read pkt5	

Assuming that node 1 has access to an accurate clock and can start the round of transmissions every 20 ms by broadcasting its packet, the detection of a packet from node 1 can also be used as a start-of-frame signal by the other nodes. On receipt of the packet from node 1, node 2 broadcasts its packet. Similarly, when node 3 has received packets from nodes 1 and 2, it broadcasts its packet, and so on. Although this is a simple protocol, the transmission of each packet is typically less than 100 μs, so that all packet transfers are likely to be completed within 1 ms leaving the rest of the frame for processing and ensuring that the bus loading is low. There are further considerations with this simple protocol:

- The data in the transmitted packet was created during the previous frame.
- One node is responsible for the synchronisation of the frames, implying the need for an accurate clock.
- The network is dedicated, that is to say, there is no other traffic on the network, which would otherwise introduce possible collisions and delays.
- All nodes follow the protocol at all times, reading incoming packets immediately and only broadcasting their packet in their allotted slot.
- If any packet is lost or corrupted, the protocol may fail and the transfers may halt, otherwise a timeout mechanism must be added to the protocol.

The protocol is totally dependent on the reliability of the network. In practice, with a dedicated Ethernet switch and modern Ethernet cards and cables, loss or corruption of data is extremely rare. In addition, in many applications, a corrupted packet could possibly be discarded without a major impact on the simulation.

From the software developer's perspective, while a distributed system offers increased speed, it adds a further layer of complexity. Specific data items used in a simulation can only be stored in one node. Within that node, the data items can be accessed directly with negligible delay. However, for other nodes needing access to these data items, the items are only visible in the transmitted packets and can only be read once a new packet has been received. These constraints place responsibility on the system designer to arrange both the functionality and the storage of data in the most appropriate nodes, in particular to reduce the number of accesses to items via packet transfers. Typically, packets are read and written by copying blocks of data rather than copying individual items.

Note that the protocol just outlined is not the only solution to the interconnection of distributed systems. Other bus systems, including PROFIBUS and CANBUS, are used for real-time applications, although bandwidth reduces with distance between nodes for CANBUS. TCP/IP may also provide a valid solution where the overall bus loading is low. Shared memory systems are also used in distributed systems which allow computers to read and write to dedicated memory shared by all the nodes, where contention for the memory is managed by the shared memory controller. However, the complexity and cost of shared memory systems tend to outweigh their advantages. Finally, it is worth noting that Airbus adapted Ethernet for the AFDX databus used for safety critical applications in the Airbus A380, the Airbus A400M and the Boeing 787 aircraft. AFDX is defined in the ARINC Specification 664, Part 7.

1.5 Processes and Threads

1.5.1 Multi-tasking

While parallelism is explicit in a distributed architecture, there may be cases within an individual computer where the computational tasks of an individual processor can also be executed in

parallel. Although an individual processor executes instructions sequentially, if the processor switches between a number of computing tasks during a single frame, it would appear that the tasks have been executed in parallel during that frame.

Consider the three examples of multi-tasking shown in Figure 1.4. In Figure 1.4(a), the three modules are executed sequentially in the order X_1, X_2 and X_3. If all three modules can complete their tasks within the frame, this is the simplest solution. Note that delays in one module will reduce the amount of processing time available to the other modules. All modules share the same code space, data regions, machine registers and system stack.

In Figure 1.4(b), the modules are arranged as processes (Hansen, 1973), which are started and stopped by the operating system and are suspended while waiting for an event. Each process has its own separate code, data, machine registers and stack space. The continuous switching of processes is undertaken by the operating system and care is needed that sufficient time is allocated to the processes to enable them to complete their tasks during each frame. The user may define the priority of processes to ensure the responses to inputs.

An alternative scheme is shown in Figure 1.4(c), which comprises three threads. Threads can be considered as lightweight processes; they are spawned by a parent thread and can share the same code and data, but each thread has its own stack and set of registers. On processors with multiple cores, threads can be allocated to cores, executing code in parallel with the other cores. From the simulation perspective, the main advantage is that threads can be designed to control the synchronisation of threads and the sharing of data. Generally, threads cooperate to complete their computations.

1.5.2 Semaphores

The Posix API enables threads to spawn new threads and to remove completed threads, provides communication between threads and ensures mutual exclusion to shared resources. Threads are particularly useful where computations need to be synchronised, or there are

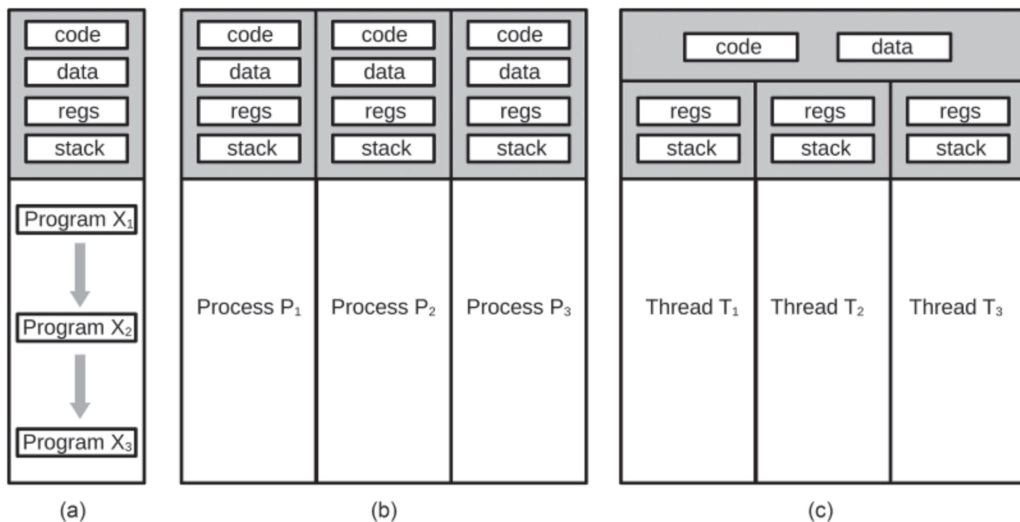


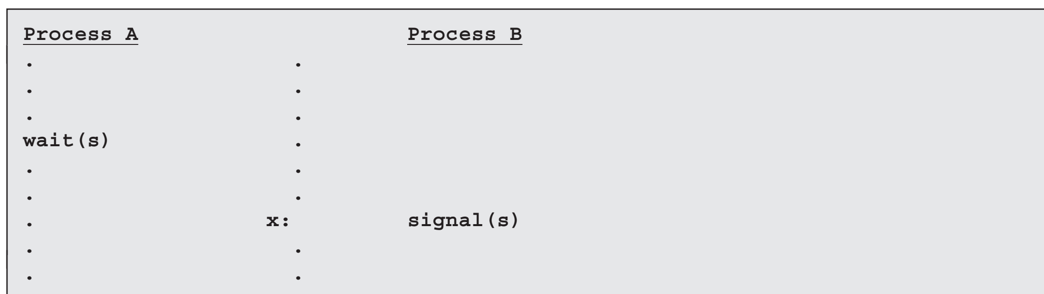
Figure 1.4 Multi-tasking Paradigms.

delays in acquiring data or where common areas of memory are shared. The communication mechanism between threads to control the sequencing of threads and accessing of shared data is provided by *semaphores*, which enable user code to control the sequencing of threads, rather than the operating system. Posix provides two functions to manage semaphores, *sem_post* and *sem_wait*, which are illustrated by the following code, where the semaphore **sem** is represented by an integer:

```
void signal(int *sem)
{
    *sem = *sem + 1;
}
void wait(int *sem)
{
    repeat
    {
        if (*sem > 0)
        {
            *sem = *sem - 1;
            return;
        }
    }
}
```

Although the code appears to be very simple, it is important to understand the significance of these operations, which are provided as Posix functions. The code in the circled sections, which can alter the state of a semaphore, must be atomic, that is, it cannot be interrupted. If an interrupt occurs while the semaphore is being altered in *signal* or *wait* and another thread subsequently modifies the semaphore, the state of the semaphore could become corrupted. This situation is avoided by the coding of *signal* and *wait* in Posix. Rather than disabling and enabling interrupts, the semaphore is modified in a single instruction, which, for many computers, is provided explicitly for that purpose. Note that, if the *wait* fails, the thread cannot continue until the semaphore is signalled by another thread, and the waiting thread is immediately suspended by the operating system.

Synchronisation is where one thread cannot proceed until another thread has reached some point in time. Initially, **s=0** and process A cannot proceed until process B reaches **x**.



```
Process A                          Process B
.                           .
.                           .
.                           .
wait(s)                     .
.                           .
.                           .
.                        x:        signal(s)
.                           .
.                           .
```

In mutual exclusion, two processes can update the information in a shared buffer without corrupting the data in the buffer, including any pointers. In Posix, the semaphore used for mutual

exclusion is referred to as a *mutex*. Initially, **s=1** and between the wait and signal calls, each process has unique access to the content of the shared buffer and can update the structure representing the buffer.

```
Process A                      Process B

wait(s)                        wait(s)
add an item to the buffer      remove an item from the buffer
signal(s)                      signal(s)
```

The producer-consumer problem (Maekawa et al., 1987) is an advanced form of a shared buffer. One thread produces items which are written to a buffer and another thread extracts the items from the buffer. Three semaphores are used. Initially, the semaphore **space = N**, defining the number of items **N** before the buffer overflows, **item** (initially 0) and **buffer** (initially 1) are two semaphores providing mutual exclusion to the item and the shared buffer, respectively. The following fragment shows the insertion and removal of one item in the buffer. If the buffer is full, no further items can be inserted until items have been removed by the consumer thread.

```
Consumer Process             Producer Process
    wait(item)                  produce an item
    wait(buffer)                wait(space)
    extract an item             wait(buffer)
    signal(buffer)              insert the item
    signal(space)               signal(buffer)
    consume the item            signal(item)
```

1.5.3 Asynchronous Input

A common example of the use of threads in simulation is where data is entered asynchronously. Simply waiting for data would introduce long delays. One solution is to have one thread acquiring input data and a second thread which only accesses the data when a complete set of data has been acquired. The following example is based on an Airbus Flight Control Unit (FCU), which is connected via an RS-232C serial line at 9600 baud (bits per second). Data is transmitted as a string of ASCII characters when a knob or a selector is turned or a button is pressed. Similarly, the computer can respond by sending an ASCII string to switch lamps on or off or to display values on an LED display. The threads interface uses the Posix interface (IEEE Std 1003, 2017) which provides a well-defined set of functions to create and manage threads. Typically, the strings contain 5–10 characters which take approximately 5–10 ms to transmit at 9600 baud. The following code avoids such delays; when no inputs have been entered, the input thread is dormant, and does not consume any frame time. When a complete string is detected, the characters are removed from the shared buffer and the buffer pointers are updated, typically taking less than a microsecond. An implementation using Posix threads (Lewine, 1991) is shown in Figure 1.5.

During initialisation:

- Two mutexs are initialised for the shared buffers.
- Two semaphores are initialised to manage the states of the threads.
- Two threads are created, one to read serial data from the FCU and one to write serial data to the FCU.

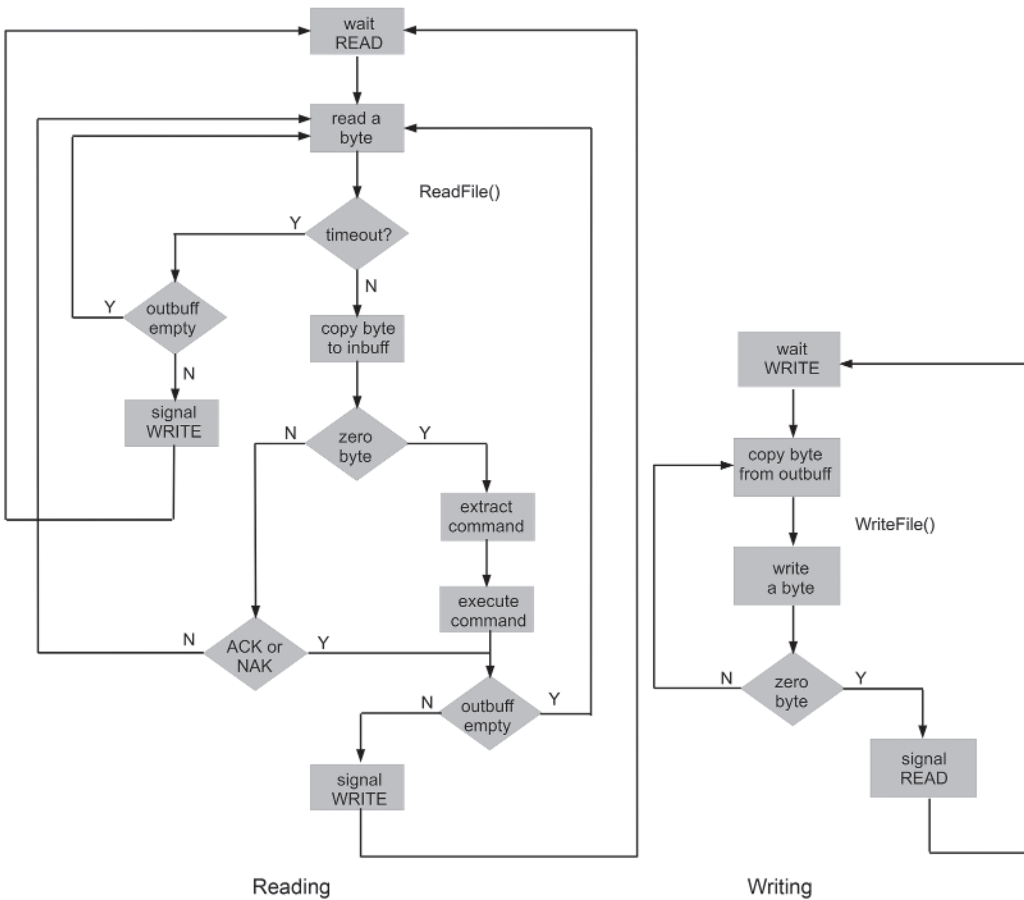


Figure 1.5 Posix Threads to Read Strings from, and Send Strings to, the FCU.

```
pthread_mutex_init(&rxbuf_mutex, NULL);
pthread_mutex_init(&txbuf_mutex, NULL);

if (sem_init(&rxsem, 0, 0) < 0)
{
    printf("sem_init failed rx thread\n");
    exit(1);
}
if (sem_init(&txsem, 0, 0) < 0)
{
    printf("sem_init failed tx thread\n");
    exit(1);
}

if (pthread_create(&port_rxthread, NULL, serial_input, NULL))
{
    printf("Unable to create serial port rx thread\n");
    exit(1);
}
```

```
if (pthread_create(&port_txthread, NULL, serial_output, NULL))
{
    printf("Unable to create serial port tx thread\n");
    exit(1);
}
```

The following code shows the thread reading data from the serial port, with pseudo-code (in italics) to simplify the code and hide unnecessary detail.

```
void *serial_input()
{
    while (1)  /* threads reads one byte from serial input indefinitely */
    {
        sem_wait(&rxsem);

        while (1)
        {
            unsigned char ch;
            DWORD         dwBytesRead = 0;

            Try to read one character from the serial port into ch

            if (dwBytesRead == 0)  /* timeout? */
            {
                pthread_mutex_lock(&txbuf_mutex);
                if (txiptr == txoptr) /* output buffer empty? */
                {
                    pthread_mutex_unlock(&txbuf_mutex);
                    continue;
                }
                else
                {
                    pthread_mutex_unlock(&txbuf_mutex);
                    sem_post(&txsem);
                    break;
                }
            }

            if (dwBytesRead != 1)
            {
                printf("serial port read error 2\n");
                exit(1);
            }

            pthread_mutex_lock(&rxbuf_mutex);
            if (!(ch ==  ACK || ch == NACK))
            {
                Enter the character into the input buffer
            }
            pthread_mutex_unlock(&rxbuf_mutex);
            if (ch == '\0') /* end of string? */
            {
                char v[100];
                unsigned int p = 0;

                pthread_mutex_lock(&rxbuf_mutex);
                while (1)
                {
                    Copy characters from the buffer to v, chx is the last character accessed
                    if (chx == '\0')
                    {
```

```
                    break;
                }
            }
            pthread_mutex_unlock(&rxbuf_mutex);
            Decode(v);
        }
        else if (!(ch == ACK || ch == NACK))
        {
            continue;
        }

        pthread_mutex_lock(&txbuf_mutex);
        if (txiptr == txoptr)
        {
            pthread_mutex_unlock(&txbuf_mutex);
            continue;
        }
        else
        {
            pthread_mutex_unlock(&txbuf_mutex);
            sem_post(&txsem);
            break;
        }
    }
  }
}
```

Although the code appears to be complicated at first sight, the use of a mutex ensures that the circular input buffer can be altered by each thread, safe in the knowledge that it has exclusive access to the buffer. The unconditional loops show that, once initiated, the thread repeats these functions, reading and decoding strings, until it is cancelled.

The function **`decode`** takes a complete string, terminated with a zero byte and, depending on the specific command, will transmit a string to the FCU in response to the command. In the decode function, as a string is decoded, the appropriate simulator value is updated. This means that the main simulator code simply reads the current values of an FCU variable unhindered by the two threads managing communication with the FCU. Apart from initialisation, all commands are initiated by the FCU, in response to a button being pressed or a knob being turned. The background threads manage the asynchronous reading of incoming strings and the transmission of strings, responding to the commands without introducing any delay during a frame.

1.5.4 Real-time Scheduling

When user processes are executing, they can only be pre-empted when a request is made for system resources. If the request fails, the process is suspended and another process is selected by the scheduler. The one exception to this scheme is the system clock, which interrupts the processor on a regular basis, ensuring a guaranteed point of pre-emption. The system hardware clock, which is not necessarily the clock maintaining the time of day, serves four very important purposes:

1) If a process has not responded for a specific time, the clock interrupt provides a means to investigate or abort a process that is behaving abnormally – this mechanism is often known as a watchdog timer. Strictly, it is the only method to detect a 'rogue' process.
2) It provides further granularity for process scheduling by providing a measure of the number of clock ticks that have occurred while a process has been running or the number of ticks that

have elapsed since a process last ran, avoiding the situation where a process is permanently locked out.

3) If no processes are running, it activates the scheduler on a regular basis, enabling the scheduler to detect if there is a new process to run.
4) With sufficient granularity, it can provide a timing reference for processes.

Note that the processing of these clock interrupts is an overhead, albeit consuming a relatively small amount of the frame time.

1.6 Software Partitioning

The modern flight simulator is likely to be based on a set of computers rather than a single fast computer, where the computers are connected to a high-speed local-area network. Given this framework, the developer is faced with the problem of organising and partitioning the software on a distributed architecture.

Data structures and their variables used in a distributed system fall into three categories: system-wide, global and local. In the overall simulation, *system-wide* variables are shared between several computers. The only mechanism to access these variables is the sharing of packets transmitted over the network. In this sense, the scope of these variables is the packet structures. For example, if the altitude of the aircraft is stored in computer A, computer B can only update its copy of altitude when the aircraft altitude is transmitted by computer A. System variables exist for the life of a program; their storage is allocated when the program is loaded and released when the program is terminated.

Within a computer, there may be several modules and each module may consist of a set of functions (or procedures). These modules may be separately compiled and linked to form an executable program. Variables shared between modules are referred to as *global* variables and, typically, these variables are defined in header files shared by the modules and declared in the module associated with the header file. A module can access these global variables by including the header file in the compilation. Typically, in a C program, these variables have the qualifier *extern*. However, if a variable does not need to be accessed by other modules but is still accessed by several functions of a module, these variables are also global variables, but without the requirement to be defined in header files. In this case, global variables are declared at the start of a program before the functions of a module are declared. Global variables exist for the life of a program; their storage is allocated when the program is loaded and released when the program is terminated.

The other group of data structures are the *local* variables. These variables are defined in a function (and include the parameters of a function) and their scope is strictly limited to the function. Specifically, variables defined in a function are not accessible to other functions. Storage for local variables is created on entry to a function and freed on exit from the function. Normally, local variables exist on a stack, which grows in one direction as variables are pushed onto the stack and reduces as variables are popped from the stack. In high-level languages, the code to manage the stack, pushing and popping variables on entry to and exit from a function, is generated by a compiler and is transparent to the user. In applications where functions are called recursively, that is to say, a function can call itself, a new set of local variables is created each time the function is invoked and deleted on each return from the function.

It is recommended practice that variables are defined in terms of minimal scope. If a variable is only used within a function, it should be a local variable of that function. If a variable is used by

two or more functions of a module, it should be defined as a global variable in the module. If a global variable is used by several modules, it should be defined in a shared header file. If a variable is used by several computers, it should be defined in a system-wide header file and accessed via a packet. Consider the example shown in Figure 1.6.

The variable x in computer A is transmitted in a packet to computer B. Module E in computer B has a copy x' of x but x is not visible to computer C. In computer B, modules D and E have access to variable y, if y is defined in a shared header file and declared in module D. y is visible to modules D and E in computer B but not to computers A and C. In computer C, z is visible to functions P, Q and R but variable w is only visible to function R. Variable z is not visible to computers A and B and variable w is not visible to functions P and Q, nor to other modules in computer C, nor to computers A and B.

The partitioning of modules to computers and functions to modules includes the data associated with the modules and functions. The partitioning of data structures and variables across computers is influenced by:

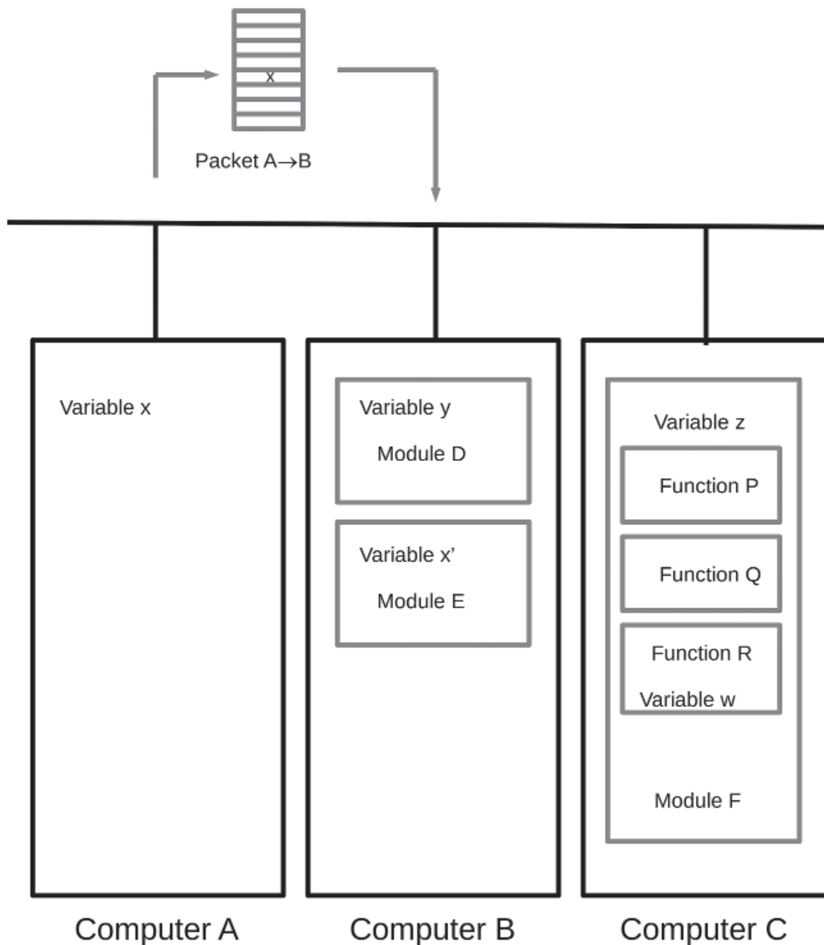


Figure 1.6 Scoping of Variables in a Distributed Architecture.

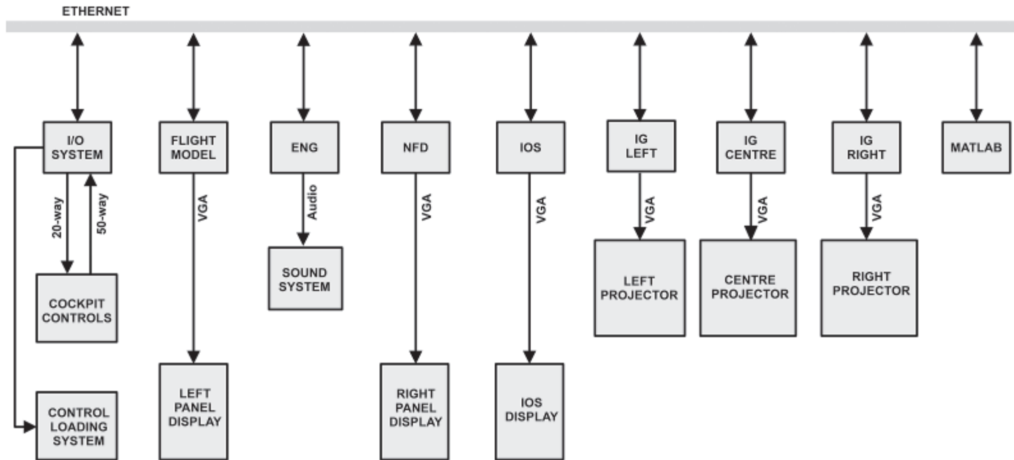


Figure 1.7 A Flight Simulator Configuration.

- the functions which need access to the variables;
- the frequency of access to the variables;
- the storage requirements of the data structures;
- clarity of the code in accessing variables.

Clearly, there is no advantage in accessing a variable in a packet if it is only used within one procedure or one module. There would be an increased overhead to form the packet and to access the variable in the packet. Similarly, if a data structure is large, unless there is an obvious need to include it in a packet transfer, data structures of this form are best restricted to one computer.

One method to partition a large project is to set out all the modules and their variables and then draw a graph with lines between modules to indicate the sharing of variables or their dependence on each other. The intensity of the interconnections is an indication of clustering and modules can be allocated to computers in order to minimise the transfer of data across the network. However, it is often likely that there is a natural partitioning of modules to computers and functions to modules, based on knowledge of the application. Consider the flight simulator configuration shown in Figure 1.7.

This example is taken from an actual engineering flight simulator, comprising nine computers, where the functions of each computer are outlined in Table 1.2. In this example, there is a clear mapping of functions to computers. Depending on the maximum frame time on each computer, some functions could be merged, provided the frame time is not exceeded.

1.7 Simulator Data

Anyone given the task of developing a flight model for a simulator is faced with an immediate problem – where is the data for the aircraft? Some basic data may be in the public domain, for example, brochures issued by the manufacturer or aviation publications (Anon, 2020). Some data may be proprietary, for example, maintenance data issued to an airline or technical manuals, in which case approval is needed to access the data. This data may include basic dimensions and performance data but is likely to lack detailed dimensions, breakdown of masses or

Table 1.2 Computer Functions.

Computer	Function
I/O	Capture digital and analogue inputs and output to the control loading system
Flight model	Aerodynamic model, gear model, FCS and equations of motion
Engine model	Engine model and sound generation
Navigation	Navigation, avionics, FCU panel and radio panels
IOS	Instructor station
IG left	IG left channel (passive)
IG centre	IG centre channel (passive)
IG right	IG right channel (passive)
MATLAB	MATLAB control laws (optional)

moments of inertia, aerodynamic data such as the coefficient of lift or examples of the aircraft response to test inputs.

The major source of data for most manufacturers is flight test data which requires instrumentation of the aircraft to measure accelerations, rates, velocities and attitude etc. The hourly rate of flying prototype aircraft to capture flight data is expensive and the flight tests need to cover the full flight envelope, possibly with hundreds of tests at specific points. Consequently, manufacturers are reluctant to release information into the public domain. In addition, the captured data is processed and organised in a format that can be accessed by a range of developers and users. It is hardly surprising that a manufacturer, investing tens of millions of dollars (and possibly more) in flight test programs, is protective of its data and would seek payment to recover these costs.

For flight simulator companies developing flight simulators for airlines and military organisations, the cost of the data packs for the aircraft, which include aerodynamic data, engine data and avionics data, is included in the price of the simulator and manufacturers are likely to charge several million dollars per aircraft for a data pack. In addition, updates to the aircraft or new data obtained for the aircraft are passed to the simulator manufacturer on a regular basis, in order to keep the simulator software up-to-date.

For other users, needing to develop flight models, but lacking funding to obtain proprietary data packs, the situation is far from clear. Assuming the aircraft is in service, it is possible to undertake flight tests but, of course, the costs of these trials is similar to the costs borne by the manufacturer and the amount of data required depends on the fidelity of the simulator. It is probably fair to say that, apart from simulator manufacturers, the cost of flight trials to acquire data can be prohibitive.

The alternatives are bleak. Data may be available in technical papers or textbooks where the authors have been given access to proprietary information. However, the drawback is that such data is likely to be incomplete. Quite simply, the manufacturer's data may occupy several thousand pages, which cannot be replicated in a chapter of a book or in a journal paper. Moreover, the data needs to include not only the aerodynamic data but also the data to validate the aerodynamic model. Not surprisingly, there are very few examples of the complete data for simulators. An exception is the large repository of NASA reports which are made available to the public via the NASA technical reports server (https://ntrs.nasa.gov). However, there are several drawbacks with using flight data from the NASA archives:

- The catalogue is not ordered in a way that enables users to locate simulation models directly.
- Many of the aircraft covered in the archive are prototype research aircraft (in line with NASA's remit).
- Most of the aircraft are pre-1980.
- Most of the flight models covered are incomplete and lack sufficient data to develop a full flight model.

In passing, it is possibly worth noting that universities and research organisations are mostly excluded from access to simulator data for modern aircraft. There are two exceptions. Firstly, an EU programme Garteur (https://garteur.org) provided data for the Airbus Beluga aircraft to participating organisations. The data is limited to the Beluga aircraft which is used to transport large aircraft components around Europe. Secondly, a generic model of a modern transport aircraft is available under the NASA Technology Transfer Program (https://technology.nasa.gov) to approved participants. The model is provided in Simulink and includes data covering all aspects of flight modelling.

There is one other source of data. Over the last 50 years, numerous papers have been published to derive aerodynamic terms of aircraft. Many of these equations are based on the geometry of an aircraft and its sub-assemblies, such as control surfaces and flaps. By assembling the papers and equations which relate strictly to the geometry of the fuselage, wing, tail, fin and control surfaces etc., it is possible to estimate aerodynamic terms from measurements of the airframe. This approach is the basis of the USAF DATCOM program (Anon, 1979), software developed at the Royal Aircraft Establishment (Mitchell, 1973), entitled 'A Computer Programme to Predict the Stability and Control Characteristics of Subsonic Aircraft' and a textbook by F. O. Smetana (Smetana, 1984) entitled *Computer-assisted Analysis of Aircraft Performance Stability and Control*. In addition, ESDU publications include the estimation of aerofoil characteristics (ESDU, 2006).

Mitchell's software was mostly limited to military organisations and the author is not aware of public domain versions of the software. Smetana's software, developed at North Carolina State University, is in the public domain and is written in FORTRAN IV and has been translated to C by the author. Smetana states clearly that the software is limited to light aircraft configurations of propeller-driven aircraft. His book contains a detailed example of a Cessna-172 aircraft and includes stability analysis but lacks details of engine modelling and undercarriage modelling. Moreover, the validation data is quite limited.

The DATCOM project was developed by the USAF in conjunction with McDonnell Douglas in 1979. It is also a large suite of FORTRAN programs but covers subsonic and supersonic regimes of flight for light aircraft, transport aircraft, military aircraft and missiles. The difficulty with both Smetana's package and DATCOM is the requirement to obtain numerous measurements of the geometry of an aircraft. In principle, once these measurements are entered into the software, it will produce overall dimensions, mass, moments of inertia and a complete set of aerodynamic derivatives. However, it does not generate data for an engine model (performance data needs to be provided) or an undercarriage model. A sample of DATCOM code for a Boeing 737-100 (Roy and Sliwa, 1983) is shown in Example 1.1.

Example 1.1 DATCOM Implementation of a Boeing 737-100

```
CASEID Boeing B-737-100
 $FLTCON WT=115000.,NMACH=1.,MACH(1)=.194,NALT=2.,ALT(1)=1500.,2000.,
         PINF=1967.62,VINF=215.68,TINF=511.57,
         NALPHA=5.,ALSCHD(1)=-2.,0.,1.,2.,4.,GAMMA=0.,RNNUB(1)=1.07E6$
 $OPTINS BLREF=93.0,SREF=1329.9,CBARR=14.3$
```

```
 $SYNTHS XW=28.3,ZW=-1.4,ALIW=1.0,XCG=41.3,ZCG=0.0,
  XH=76.6,ZH=6.2,
  XV=71.1,ZV=7.6,
  XVF=66.2,ZVF=13.1,
  VERTUP=.TRUE.$
 $BODY NX=14.,
       BNOSE=2.,BTAIL=2.,BLA=20.0,
  X(1)=0.,1.38,4.83,6.90,8.97,13.8,27.6,55.2,
        65.6,69.0,75.9,82.8,89.7,90.4,
  ZU(1)=.69,2.07,3.45,4.38,5.87,6.90,8.28,
        8.28,8.28,8.28,7.94,7.59,7.50,6.9,
  ZL(1)=-.35,-1.73,-3.45,-3.80,-4.14,-4.49,-4.83,
        -4.83,-3.45,-2.76,-0.81,1.04,4.14,6.21,
  R(1)=.34,1.38,2.76,3.45,4.14,5.18,6.21,6.21,
        5.87,5.52,4.14,2.76,.69,0.0,
  S(1)=.55,8.23,28.89,44.31,65.06,92.63,127.81,
       127.81,108.11,95.68,56.88,28.39,3.64,0.11$
 $WGPLNF CHRDR=23.8,CHRDTP=4.8,CHRDBP=12.4,
       SSPN=46.9,SSPNOP=31.1,SSPNE=40.0,CHSTAT=.25,TWISTA=0.,TYPE=1.,
       SAVSI=29.,SAVSO=26.0,DHDADI=0.,DHDADO=4.$
 $JETPWR NENGSJ=2.0,JEVLOC=-5.2,JIALOC=34.5,JELLOC=15.9,JEALOC=58.0,
         JINLTA=13.4,AIETLJ=-5.$
 $VTPLNF CHRDR=15.9,CHRDTP=4.8,SAVSI=33.,
         SSPN=27.6,SSPNOP=0.,SSPNE=20.7,CHSTAT=.25,TWISTA=0.,TYPE=1.$
 $HTPLNF CHRDR=12.4,CHRDTP=4.1,
         SSPN=17.6,SSPNE=15.87,CHSTAT=.25,TWISTA=0.,TYPE=1.,
         SAVSI=31.,DHDADI=9.$
 $SYMFLP FTYPE=1.,NDELTA=9.,DELTA(1)=-40.,-30.,-20.,-10.,
         0.,10.,20.,30.,40.,SPANFI=0.,SPANFO=14.,CHRDFI=1.72,
         CHRDFO=1.72,NTYPE=1.0,CB=.50,TC=.44,PHETE=.003,PHETEP=.002$
NACA-W-4-0012-25
NACA-H-4-0012-25
DERIV RAD
```

A fragment of the output for a Cessna Citation II aircraft showing several aerodynamic derivatives is given in Table 1.3, which is taken from the spreadsheet output generated by DATCOM. Note that this data is for one specific flight condition in terms of airspeed, altitude and engine setting with the angle of attack varying from $-2°$ to $16°$. These tests would be repeated for the complete flight envelope enabling aerodynamic derivatives to be computed as a function of Mach number and possibly altitude. The output produced by DATCOM can be specified with the input data and several packages have modified the user interface to DATCOM, providing outputs in spreadsheet and MATLAB formats and graphical form using gnuplot.

Software models developed in the NASA LarcSIM (Jackson, 1995) project, JSBSim (http://jsbsim.sourceforge.net) and FlightGear (https://www.flightgear.org), have used data derived from DATCOM. Nevertheless, both the DATCOM package and Smetana's software are not without their problems:

- The user interface is cumbersome and complicated, although a few packages have been developed to simplify the design of user interfaces.
- The output is not in a format that can be used directly in flight modelling.
- The accuracy is variable – in cases where the results are compared with flight data, inaccuracies of 10–20% are reported (Ahmad et al., 2021).

Table 1.3 Aerodynamic Data for a Cessna Citation II.

Alpha	Cladot	CYbeta	CYp	Clbeta	Clp	Clr	CMq	CMadot	CNbeta	CNp	CNr
−2	2.3619	−0.7503	−0.0879	−0.136	−0.4558	0.0295	−16.836	−6.715	0.0573	−0.0002	−0.0984
0	2.405	−0.7503	−0.0965	−0.1301	−0.4649	0.0615	−16.836	−6.8375	0.0573	−0.0154	−0.1002
2	2.486	−0.7503	−0.1055	−0.1242	−0.4724	0.0941	−16.836	−7.0678	0.0573	−0.031	−0.1024
4	2.5321	−0.7503	−0.1147	−0.1182	−0.4695	0.127	−16.836	−7.1989	0.0573	−0.0471	−0.1051
8	2.3822	−0.7503	−0.133	−0.1062	−0.3793	0.1897	−16.836	−6.7729	0.0573	−0.0824	−0.1115
9	2.2906	−0.7503	−0.1362	−0.1033	−0.3392	0.2016	−16.836	−6.5124	0.0573	−0.0908	−0.113
10	2.11	−0.7503	−0.1388	−0.1005	−0.2957	0.2118	−16.836	−5.9988	0.0573	−0.0992	−0.1145
11	1.7501	−0.7503	−0.1408	−0.0977	−0.2473	0.2203	−16.836	−4.9758	0.0573	−0.1073	−0.1157
12	1.4248	−0.7503	−0.142	−0.095	−0.194	0.2269	−16.836	−4.0509	0.0573	−0.1149	−0.1168
13	1.1986	−0.7503	−0.1421	−0.0924	−0.1476	0.2313	−16.836	−3.4078	0.0573	−0.1214	−0.1176
14	0.6856	−0.7503	−0.1437	−0.0898	−0.046	0.2342	−16.836	−1.9491	0.0573	−0.1297	−0.1183
15	−0.2414	−0.7503	−0.1897	−0.0874	0.3134	0.2309	−16.836	0.6862	0.0573	−0.1346	−0.1182
16	−0.7853	−0.7503	−0.0471	−0.0861	0.8667	0.2065	−16.836	2.2327	0.0573	−0.2453	−0.1159

- The learning curve is large – both packages come with extensive documentation but it is necessary to understand the formats and naming conventions.
- It is very easy to make mistakes in data entry and these are not always detected or flagged by the software. If the entered data is incorrect, the output will contain errors, but without validation data it is difficult to locate the sources of errors in the data.
- It is easy to make mistakes in terms of units, for example, confusing metres with feet or degrees with radians.

Despite these reservations, an understandable viewpoint is that slightly inaccurate modelling data is better than no data and, for many investigations, a representative flight model may be acceptable. Certainly, the widespread use and acceptance of DATCOM is an indication of its merit in flight modelling. The documentation provided in both the DATCOM user guide (Anon, 1979) and Smetana's textbook provides full details of the equations used in the software and the various sources used to derive the equations. Users have to decide if the considerable effort to produce aircraft geometric data in the rigid formats required by the packages is worth the benefit of developing models that are otherwise unavailable in the public domain.

1.8 Input and Output

1.8.1 Data Acquisition

A range of devices can be connected to a computer via interfaces which include serial ports, parallel ports, Universal Serial Bus (USB) ports and Integrated Development Environment (IDE), Small Computer System Interface (SCSI), and Serial Advanced Technology Attachment (SATA) hard drive interfaces. The commonality of all these interfaces is that they include registers to access the status of a device, to set up transfers and to transfer data to or from a device. Generally, users are inhibited from accessing hardware registers and transfers are initiated by system calls to the operating system, which is responsible for the transfers, in order to ensure the integrity of transfers.

Consider an example of a serial port reading data at 9600 baud (bits per second). This data rate is approximately 1000 bytes per second or one 8-bit byte per ms, or 20 bytes per frame at 50 fps. Having read one byte from the serial port, the next byte will not be available for approximately 1 ms. There are three options:

1) The processor can simply wait for the next byte to become available, typically by testing a status bit, and then reading the data. This method is known as blocking I/O and introduces the problem that the processor performs no useful operations waiting for a slow transfer. Moreover, if no data is generated, the processor could wait indefinitely.
2) The processor can check to see if a byte is available and, if so, read it, otherwise it can return an error code to indicate that no data is currently available. This method is known as non-blocking I/O and enables the processor to check for data but continue with other processing when no data is available. The processor wastes valuable time checking for data that is not available.
3) The device can be enabled to generate an interrupt when data is available. Once set up, the processor can perform other computations and will only be interrupted when the data is available. The only overhead is stopping to respond to the interrupt, capturing the data and resuming a previous activity.

A similar problem occurs with disk transfers, which have fast data rates but typically are used to transfer large amounts of data in a single transfer. While data is transferred between the disc and memory, which could take several milliseconds, the processor can continue with other processing until the transfer is completed. The method used is known as DMA. The processor accesses the device registers to specify the disc address, the memory address, the size of the transfer and the type and direction of the transfer and initiates the transfer. On completion of the transfer, the device interrupts the processor, which checks on the status of the transfer.

1.8.2 Digital-to-Analogue Conversion

Digital-to-analogue (D/A) conversion enables variables stored in a computer to be output as analogue voltages to drive electrical equipment, such as lamps, motors and actuators, for example, an autothrottle. Figure 1.8 shows a typical D/A configuration.

D/A conversion is relatively straightforward and very fast. The processor writes to the control register to configure the D/A converter, to select a channel and to write a value to be converted. The status register enables the processor to detect if the D/A is busy or if a conversion is completed and to check if any errors occurred in the conversion. The advantage of a D/A is that the bits of an output register are generated by a ladder of resistors, where each resistor has a weighted value that determines the current through the resistor. The currents are added (electrically) and converted to a voltage, as shown in Figure 1.9, which illustrates a 10-bit D/A.

The resistor values produce a current which is weighted to correspond to the bit value of the 10 bits (Horowitz and Hill, 1980). The device is relatively simple, containing 22 resistors and an operational amplifier used for summation. As only two resistor values are used, it is possible to manufacture the device to a very high tolerance. The conversion time depends on the time to write a

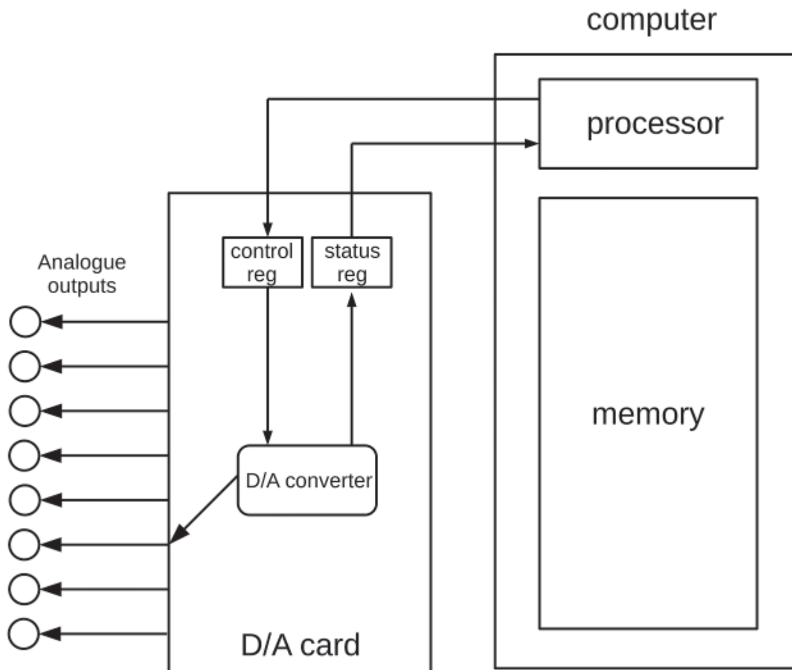


Figure 1.8 Digital-to-Analogue Conversion.

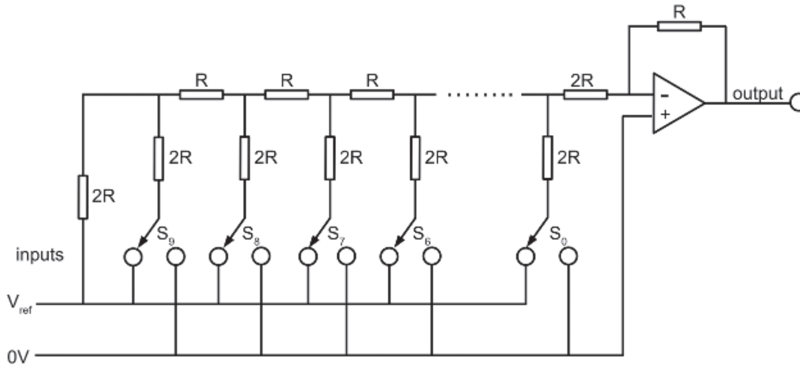


Figure 1.9 A Typical Digital-to-Analogue Converter.

10-bit value to a register and the settling time of the operational amplifier, which is typically less than a microsecond. Extending the device for higher resolutions merely requires the addition of further stages. The output of D/A devices is typically limited to a few milliamps and additional amplification is required to drive lamps and motors.

1.8.3 Analogue-to-Digital Conversion

Flight simulator inputs from the controls, levers, selectors and knobs are connected by potentiometers, LVDT transducers, strain gauges or other forms of sensors, where the signal is an analogue voltage in a defined range. The process of analogue-to-digital conversion (A/D) converts analogue inputs to digital values for use in a computer using dedicated hardware to acquire and convert these signals, which can be read directly by a computer. For A/D conversion, there are several considerations:

- the range of voltage of the inputs – if the input signal is too small, there is a potential loss of resolution and if the signal is too large, it may cause damage to the input circuitry;
- the resolution of the conversion – the number of bits used to represent the voltage;
- the time to convert the input and read it into the computer;
- the linearity of the conversion;
- the filtering of any noise in the signal.

Typically, an A/D converter is interfaced to a computer as shown in Figure 1.10.

The processor writes to the control register to select the input channel, to specify the type of transfer and to initiate the transfer. The input register enables the processor to read the sampled value and to check the status of the A/D, for example, if it has completed the conversion or if an error occurred during conversion. In this example, only eight analogue inputs are shown and they can be switched or connected to the A/D converter. The main limitation with this solution is that the processor is involved with the transfer and has to poll the A/D to detect that the conversion is completed. Alternatively, the A/D may be configured to interrupt the processor on completion of the conversion, to avoid polling. Depending on the A/D device, the number of input channels can vary from 1 to 32 (or more), the resolution can be from 8 to 16 bits and the conversion time can range from a few microseconds to several milliseconds. An improvement is shown in Figure 1.11.

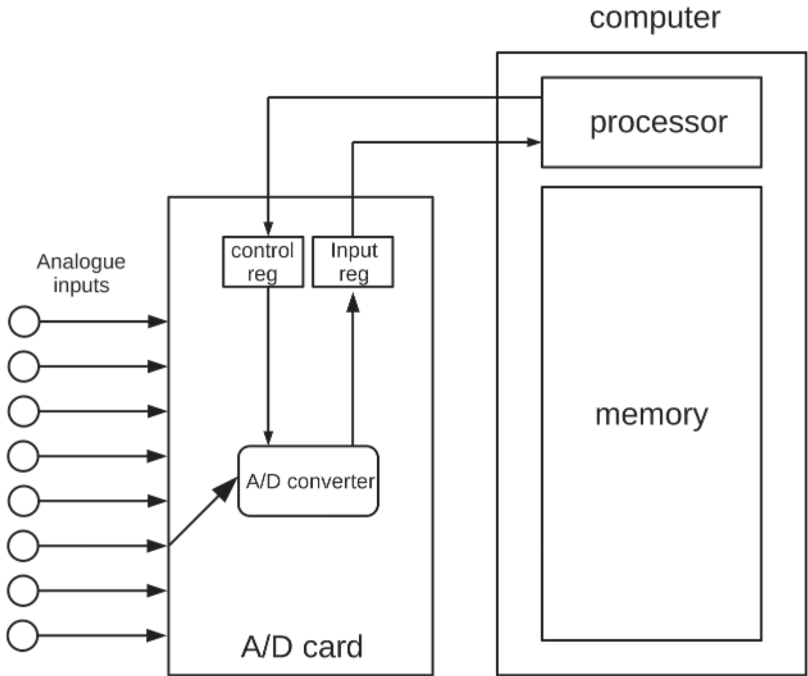


Figure 1.10 Analogue-to-Digital Conversion.

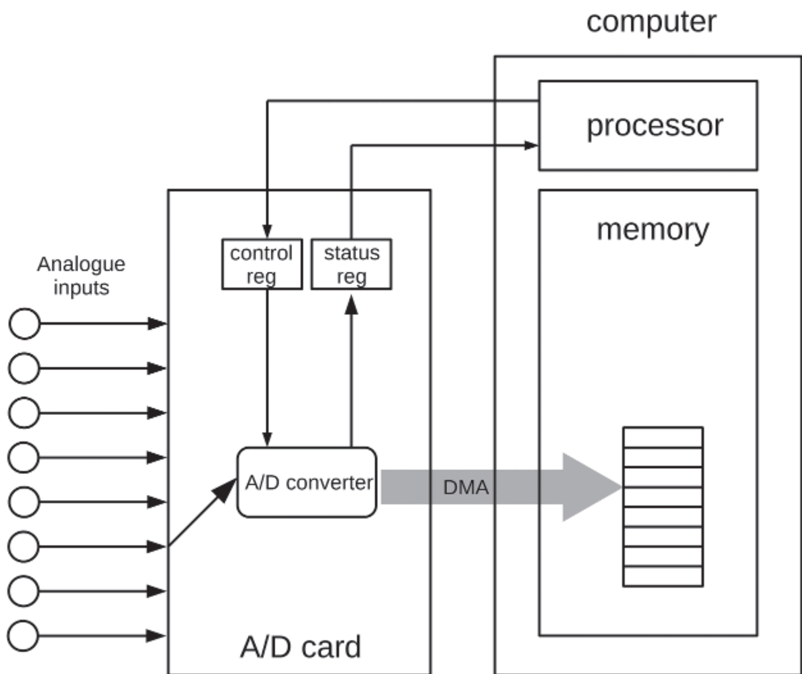


Figure 1.11 Analogue-to-Digital Conversion Using DMA.

Again, only eight analogue inputs are illustrated. The A/D converter outputs are mapped to corresponding memory locations in the processor memory using DMA transfers, so that the processor loading is limited to setting up the A/D transfers. Often, the inputs are sampled autonomously and repetitively, requiring no further intervention by the processor. The sampled values can then be accessed from memory by the simulation software. Normally, the analogue inputs are sampled at the simulator frame rate.

Unlike the D/A, extending an A/D for higher resolution is not straightforward. Three forms of A/D devices are used, with the performance depending on the A/D method:

1) The simplest and slowest form is a counter with a D/A which is initially set to zero. As the counter increments, the voltage output from the D/A rises linearly. The A/D input and the output from the D/A are connected to a comparator which saturates when the D/A ramp voltage exceeds the input voltage, at which point the counter is inhibited. The value in the counter corresponds to the input voltage.
2) A variation is to search for the input voltage by setting or clearing bits in the D/A until the difference between the D/A output and the input is within the voltage corresponding to the least significant bit of the register.
3) The fastest but most expensive device is a flash A/D. The input voltage is applied to a ladder of resistors and operational amplifiers. For n-bit conversion, 2^n stages are required. The 2^n outputs of the operational amplifier are treated as digital inputs to an n-bit encoder to produce an n-bit output. Note that a 16-bit A/D would contain over 64,000 operational amplifiers.

The sequential A/D is slowest as the conversion time depends on the rise time of the ramp voltage, whereas the successive approximation method is guaranteed to convert the input in n cycles for an n-bit A/D and is therefore significantly faster. The flash A/D is the fastest device as it only depends on the settling time of 2^n operational amplifiers, plus the small delay of the encoding; flash A/D is widely used for applications requiring high conversion speeds. However, at frame rates of 50 Hz, with 32 analogue channels, the sampling rate is only 1600 samples per second, which is well within the performance of most A/D devices.

1.8.4 Multiplexing

Although the sampling rates in real-time simulation are relatively low, many flight simulators may have several hundred analogue inputs. Rather than allocating an A/D to each channel, a single high-speed A/D can be used, where the inputs are selected by switching the multiplexor output, as shown in Figure 1.12

The configurations are shown with only eight inputs, for purposes of illustration. In Figure 1.12(a), all A/D devices are selected and the converted values are written, either to local memory (as shown) or memory-mapped in the host computer. In Figure 1.12(b), the multiplexor is addressed to select one of the eight inputs which is converted and read by the host computer.

1.8.5 Encoders

Most potentiometers used for analogue input have a fixed mechanical range. However, there are applications, particularly where the input is derived from a shaft, where the input is angular and continuous. Some potentiometers have a circular track with a small band of insulating material

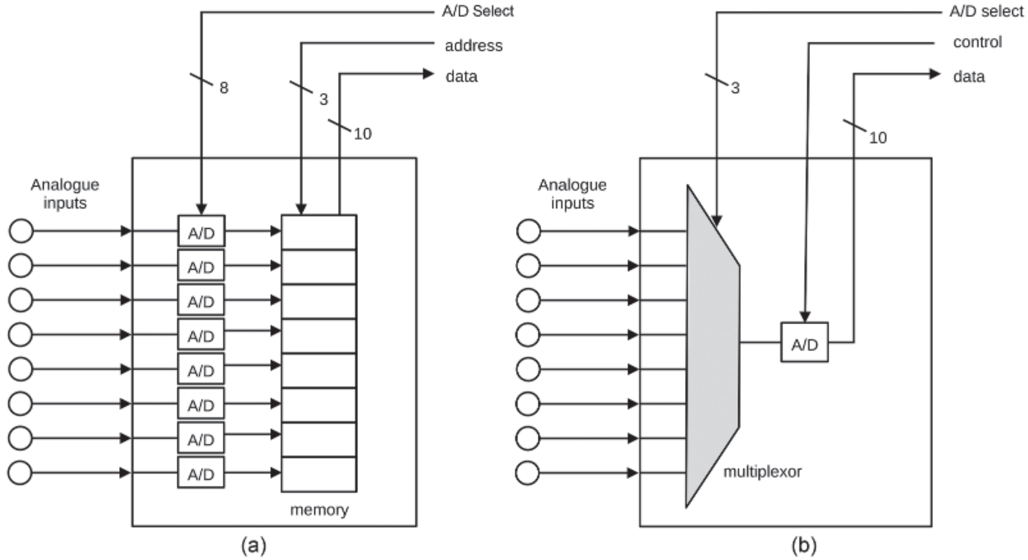


Figure 1.12 Multiplexed Analogue-to-Digital.

between the end points of the track. For example, the track of a potentiometer may range from 1 to 10 KΩ with an infinite resistance when the wiper is positioned over the insulator. An alternative method of measuring continuous angular position, for example, a radio panel knob, is to use a quadrature encoder. The encoder provides two square waveforms, A and B, which enable both motion and the direction of motion to be detected, as shown in Figure 1.13.

The signal is detected by a quadrature decoder where the inputs are A and B and the two digital outputs are the direction and a count enable to detect a change in position. These outputs can be fed to an up–down counter, as shown in Figure 1.14.

Although this arrangement requires one counter per channel, typically with 10–16 bits, it is also possible to detect the change in position and direction of motion in software, requiring only the decoder outputs. The advantage of using quadrature encoders is that the computer input is digital rather than analogue and that the cost of decoders is typically a few dollars. Quadrature decoders are used where the input is a rotary selector or knob, as they avoid additional analogue channels and the ambiguity of detecting an input where the potentiometer is positioned on the insulator.

1.8.6 Digital Input/Output

Digital inputs are used for inputs that can only take the value 0 or 1 (on or off). It is clearly extravagant to allocate an analogue channel to a digital input that can have only two voltages. Similarly, digital outputs for lamps, which are also either 0 or 1, can be represented as digital values rather than analogue values. Typically, an input/output (I/O) card will include digital input and output registers for 16 or 32 bits, or possibly more. The outputs may drive relays or amplifiers as they are likely to be limited to a few milliamps.

The digital inputs of an I/O card will be limited to a voltage range specific to the card. The most common form of input in a simulator is a switch or selector. Rather than providing an active input

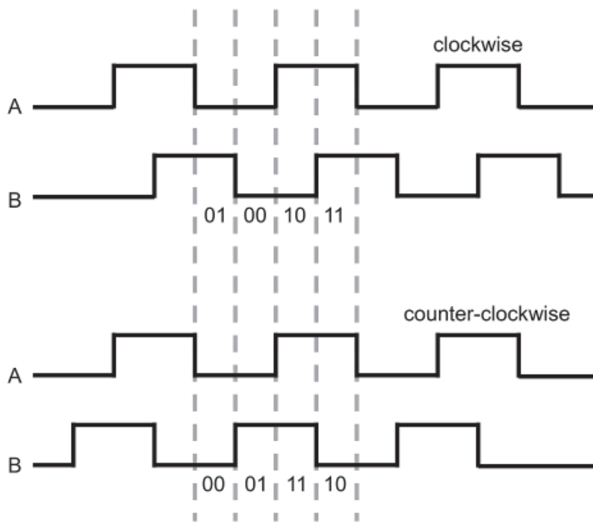


Figure 1.13 Quadrature Encoder Signals.

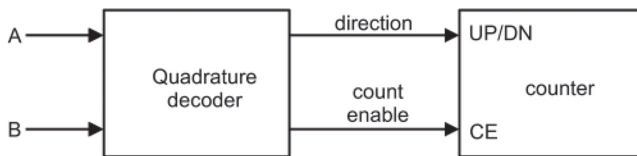


Figure 1.14 Quadrature Decoder.

that switches between 0 V and 5 V, a circuit similar to the one shown in Figure 1.15 is used for digital inputs.

If the switch is open, V_{out} is at the reference voltage V_{cc}. If the switch is closed, V_{out} is at 0 V. The value of the resistor R is chosen so that the circuit only takes a few micro amps when the switch is closed. The main advantage of this arrangement is that if the switch is accidentally grounded, no electrical damage will ensue, avoiding the situation where the supply rail could be connected accidentally to ground.

1.8.7 Signal Conditioning

Depending on the technology of I/O devices, the input range of analogue inputs may be fixed at ±10 V, ±5 V, 0–3.3 V, 0–5 V or even 0–24 V. A problem arises if the inputs are outside these ranges, or alternatively, if they only occupy a small part of a range. In the first case, protection is required to avoid damaging the input circuitry and the signal must be attenuated to the input range. In the other cases, the inputs will only use a fraction of the resolution available, effectively reducing the accuracy of the A/D.

Signal conditioning is provided with analogue inputs to ensure that the inputs extend over the full input range. For example, if a potentiometer is connected to measure the rudder position, but the

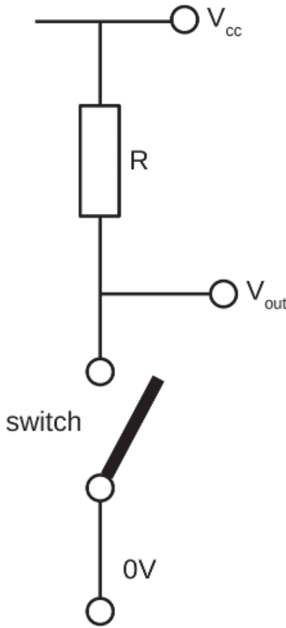


Figure 1.15 Digital Input.

mechanical movement of the assembly only displaces the potentiometer over 40% of its travel, for an input range 0–5 V, it may be found that the left full rudder produces 1.5 V and right full rudder produces 3.5 V. The inputs need to be calibrated to give 0 V for left full rudder and 5 V for right full rudder. Calibration of both the gain and offset provides bias compensation, as shown in the circuit in Figure 1.16 for each analogue input.

The bias is altered by adjusting the variable resistor R_3 where the voltages ±V are sufficient to ensure the input bias covers the range of possible input values. The gain of the circuit is adjusted by changing the ratio of R_2/R_1 where R_2 is a variable resistor. For each input, the gain and bias are adjusted so that the minimum input is 0 V and the maximum output is 5 V. As the two adjustments affect each other, calibration is an iterative process and is amenable to automation where the voltages between R_5 and R_6 and between R_2 and the negative input of the operational amplifier are provided by D/A outputs. The calibrated values can then be stored and reset at the start of the simulation. The calibration process should be undertaken as part of regular maintenance or following replacement of sensors, partly to ensure the input is correctly calibrated but also to check the polarity of the inputs.

One further consideration, particularly for analogue inputs, is electrical noise. Generally, at low sampling frequencies, there is no likelihood of radio interference or cross-talk from cabling although a circuit board can contain tracks that act as an antenna inducing noise in circuits in close proximity to electrical motors. Noise is mostly picked up from two sources, firstly, mains pick-up, where some component of the AC mains signals leaks into the circuitry and, secondly, poor grounding where there is a small voltage difference between the 0 V lines of different components.

In addition to the 0 V line there is usually a ground line with the cable shield connected to ground. As the currents into an operational amplifier are extremely small, the grounding input of all amplifiers should be common. However, if these connections occur over a relatively long distance (several metres), these grounding points may not all be at the same voltage. Moreover, the 0 V line of digital inputs and analogue inputs should be separated, otherwise interference can occur if these are connected. With 5 V inputs, the least significant bit of a 12-bit analogue input is only 1 mv and, consequently, noise levels above 500 μv are likely to corrupt the input signal. If noise is detected, there are two solutions. Firstly, the grounding of digital and analogue inputs should be checked, isolating (if possible) 0 V lines selectively, in order to identify possible sources of poor grounding. Secondly, a low-pass filter can be added to noisy channels to remove components above 100 Hz. The bandwidth of the filter must be selected to eliminate noise without filtering the actual input signal.

1.8.8 Embedded Systems

Following the development of Arm chipsets for embedded systems (Sloss et al., 2004), several vendors developed small footprint devices including the Raspberry Pi (RPi) and the BeagleBone single-board computers. These devices include USB ports, Ethernet connection, HDMI ports, I/O

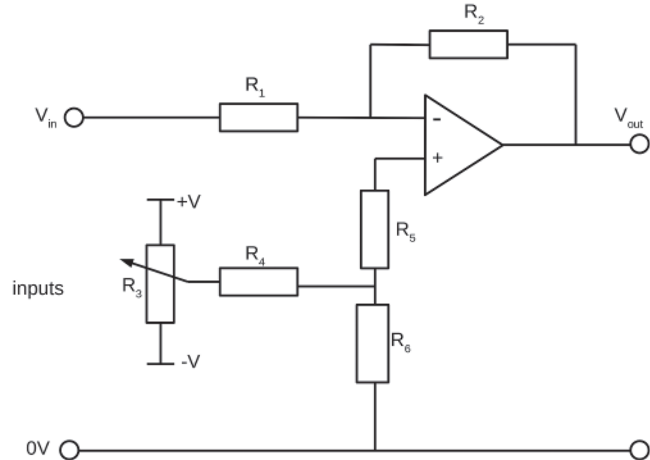


Figure 1.16 Analogue Input Conditioning.

pins and SD-card storage. In addition, there is sufficient processing power and memory capacity to run a Linux operating system. One option is to use these devices as headless embedded systems, where the code is developed to run automatically at power-up in a dedicated system. One other important aspect of these developments is that the cost of an RPi is of the order of $40. With the performance similar to a low-end PC and 40 I/O pins, the RPi can be used as a dedicated I/O interface for real-time simulation.

With the addition of A/D, D/A and digital I/O integrated circuits, the RPi can provide a dedicated controller for analogue and digital I/O at the simulator frame rate. In addition, acquired data can be transmitted in an Ethernet packet, synchronised to the start of every frame, which is read by all the computers connected to the network. The RPi can perform one further function which is the provision of frame timing. The Linux clock is accurate to within a few microseconds and the transmission of a packet containing the system I/O data can signal the start of the simulator-wide frame.

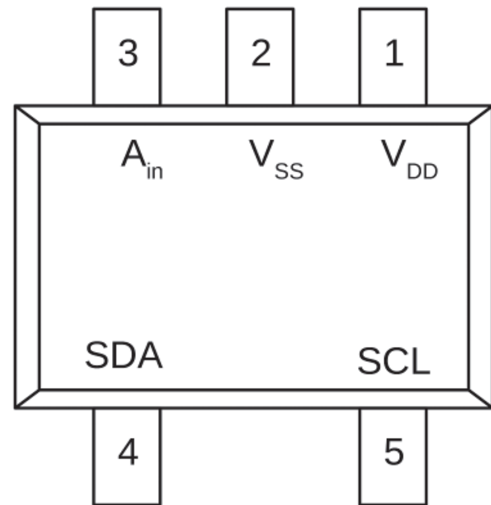


Figure 1.17 An I^2C A/D.

The RPi has a 40-pin header known as the general purpose I/O (GPIO) line. This feature makes the RPi well-suited to hardware applications needing an embedded processor. The I^2C (I squared C) protocol was originally developed by Philips Semiconductors in 1982 (Anon, 2012) and has become a worldwide standard for low-cost interfacing. The attraction of I^2C is that it only requires four wires, including the supply reference voltage (V_{DD}) and ground (0 V). The other two wires, SDA and SCL, provide the timing, control and data as serial signals between the processor and a device. I^2C supports data rates of 100 Kbit/s and 400 Kbit/s. Readers seeking to learn more about I^2C signal formats and protocol are referred to the extensive literature on the subject.

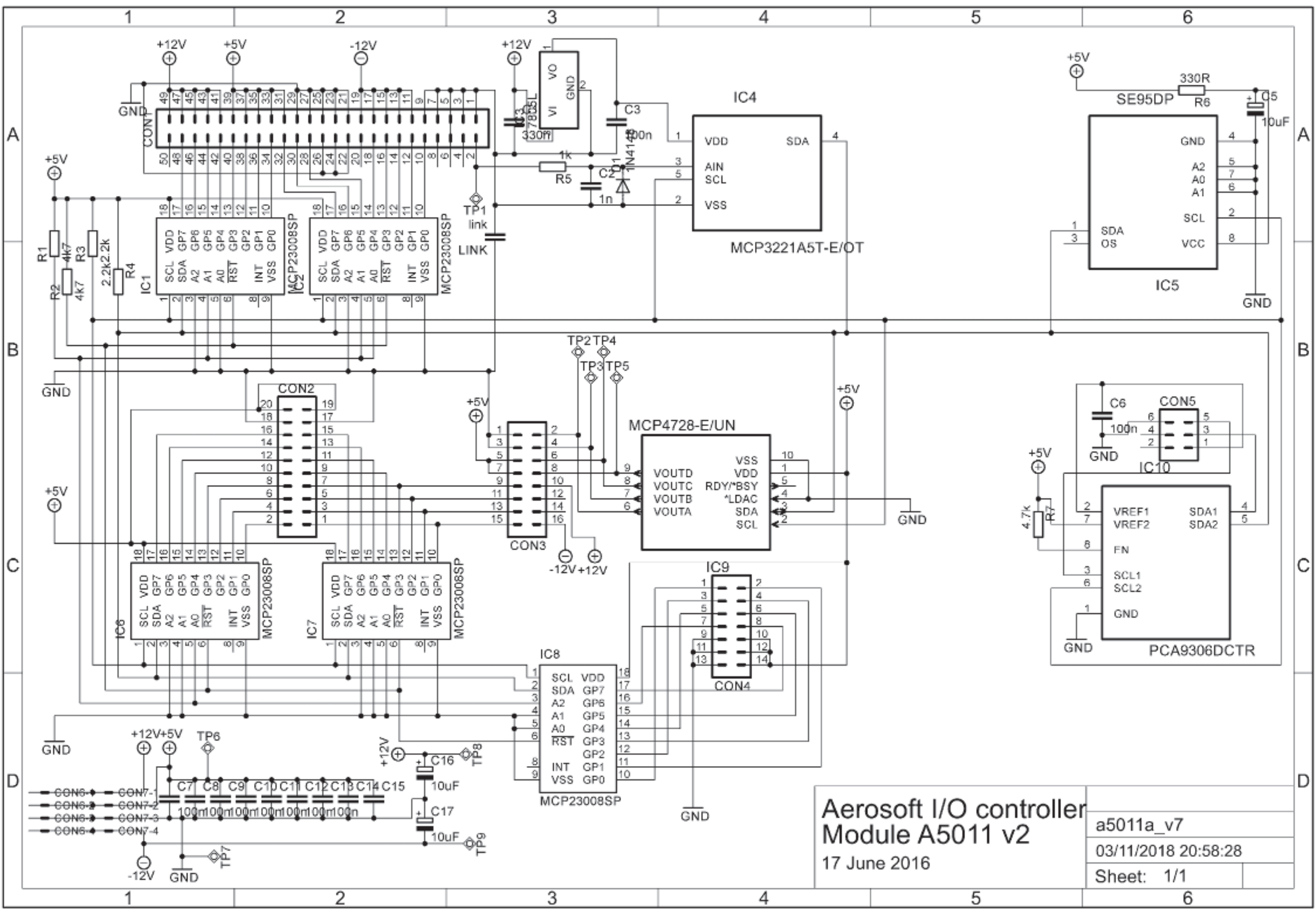


Figure 1.18 An I^2C I/O Interface for a Raspberry Pi.

With only four wires between the RPi processor and an I^2C device, the hardware interfacing is remarkably straightforward. Many vendors sell a wide range of I^2C devices which include motor control, A/D, D/A, serial ports, parallel ports, microcontrollers and so on. For example, the MCP-3221, shown in Figure 1.17, is a 12-bit analogue-to-digital converter manufactured by Microchip Technology Inc (https://www.microchip.com/en-us/product/MCP3221).

Available in surface-mount form, the integrated circuit is approximately 3 mm × 3 mm, V_{DD} is the reference voltage, V_{SS} is 0 V, SDA and SCL are the I^2C lines and A_{in} is the analogue input. A slight complication is that the I^2C and the RPi reference voltages are 5 V and 3.3 V, respectively; an additional chip is used to convert V_{DD}, SDA and SCL voltage levels between the RPi and the I^2C chips. The schematic of a small board containing an A/D for 32 12-bit multiplexed analogue inputs, four 12-bit D/A outputs, 32 digital inputs, and 16 digital outputs is shown in Figure 1.18. The MCP-23008 dual in-line packages are used for digital input and output. Note that, in addition to the four I^2C lines, there are three address lines for each IC, enabling up to eight MCP-23008 ICs to be addressed separately. Five of the digital outputs are used to drive two external multiplexors for the 32 analogue inputs.

The simplicity of this solution for an I/O device is remarkable. There are only 10 ICs on the board with minimal wiring owing to the I^2C serial protocol. The MCP-3221 A/D sampling rate is over 22,000 samples per second. The MCP-4728 D/A has a settling time of 6 μs. The MCP-23008 I/O expander provides 8-bit digital inputs and outputs and allows the direction of each bit to be set independently.

The programming of I^2C devices is straightforward. The I^2C protocol must be enabled and the *i2c-tools* package, which can be downloaded for most distributions, provides the drivers for RPi I^2C devices. The code to sample a single analogue input channel is shown here, to illustrate the basic principles of transferring data via I^2C devices:

```
void Sample_ADC(unsigned int chn)
{
    unsigned char                inbuf[2];
    unsigned char                outbuf[2];
    struct i2c_rdwr_ioctl_data packets;
    struct i2c_msg               messages[2];

    outbuf[0] = 9;  /* reg 9 = channel no for analogue mux */
    outbuf[1] = (unsigned char) chn;

    messages[0].addr = MUX_ADR;
    messages[0].flags = 0;
    messages[0].len = 2;
    messages[0].buf = outbuf;

    packets.msgs = messages;
    packets.nmsgs = 1;

    if (ioctl(i2c, I2C_RDWR, &packets) < 0)
    {
        printf("Unable to set the MUX dir register\n");
        exit(1);
    }
    messages[0].addr = ADC_ADR;
    messages[0].flags = I2C_M_RD;
    messages[0].len = 2;
    messages[0].buf = inbuf;
```

```
    packets.msgs = messages;
    packets.nmsgs = 1;

    if (ioctl(i2c, I2C_RDWR, &packets) < 0)
    {
        printf("unable to read ADC ch=%d\n", chn);
        exit(1);
    }
    ADC[chn] = (((unsigned int) inbuf[0] & 0xf) << 8) + (unsigned int) inbuf[1];
}
```

Each MCP-23008 has a unique address and **MUX_ADDR=0x23**. The system function **ioctl** is used to access the I^2C devices with three arguments: the file descriptor, the I/O request and a pointer to the data. The first **ioctl** call sets the multiplexor to one of the 32 channels. The second **ioctl** call reads the 12-bit data from two registers (**inbuf**) which is written to an array holding 32-bit values. Note that the **ioctl** calls are checked and if errors are detected, the software is terminated, as further I^2C transfers are likely to fail. In this example, the initialisation of the I^2C devices is omitted.

The use of a dedicated embedded system for I/O transfers has many advantages. Although the I^2C polling is an overhead for the RPi, it can easily sustain the frame rate. For the other computers, the overhead is one packet transfer; the raw data sampled by the RPi is read in a small Ethernet packet broadcast by the RPi at the start of each frame. The I/O board is sparsely populated and approximately three inches square with a small connector to four GPIO pins. The main limitations with I^2C are that it is only possible to directly address 128 I^2C devices, the data rate is nominally limited to 400 Kbit/s and the interconnection distance is limited to a few feet. However, none of these limitations apply to real-time flight simulation and expansion of the I/O system using additional boards or additional RPi computers is straightforward.

1.8.9 USB Interfacing

In recent years, many vendors have interfaced their equipment via USB ports rather than analogue or digital inputs, partly because USB ports are commonly provided on modern computers but also because USB simplifies the I/O interface. The technology of USB has advanced, with data rates for version 1.0 of 1.5 Mbits/s increasing to 5 Gbits/s with version 3.0. The acquisition of analogue and digital inputs is implemented in external hardware, reducing the loading on the host computer to acquire input data.

Although drivers may be available to interface to commercial equipment, Linux provides generic USB interface libraries; for example, a system header file **<linux/joystick.h>** is used for access to games devices via USB ports. One example of simulator hardware is the ThrustMaster™ set of controls, requiring three USB inputs for the side-stick, rudder and throttle inputs. The following code opens an input channel for the first USB port:

```
js0 = open("/dev/input/js0", O_RDONLY | O_NONBLOCK);

if (js0 < 1)
{
    printf("Unable to open joystick\n");
    exit(-1);
}
```

The USB input is opened for reading from port **/dev/input/js0** in a non-blocking mode, that is, incoming data can be read without waiting for the data to be generated. If data is available, it can be read, otherwise the code returns immediately, with a status indicating that no data is available. Linux system **ioctl** calls are used to access USB ports. For example, to check the initial status of a side-stick, the following code obtains the name of the device, its version number, the number of axes and the number of buttons.

```
ioctl(js0, JSIOCGNAME(sizeof(name)), name);
ioctl(js0, JSIOCGVERSION, &version);
printf("%s: version %d\n", name, version);
ioctl(js0, JSIOCGAXES, &naxes);
printf("%d axes\n", naxes);
ioctl(js0, JSIOCGBUTTONS, &nbuttons);
printf("%d buttons\n", nbuttons);
```

One reason for accessing the device name is that control inputs may be connected arbitrarily to the USB ports; identifying the device name enables a dynamic mapping to be used between the device channels and USB ports, to ensure that the correct channel is accessed.

The main function is to read characters from the device and this is implemented in the following code:

```
int read_event(int fd, struct js_event *event)
{
    ssize_t bytes;

    bytes = read(fd, event, sizeof(*event));

    if (bytes > 0)
    {
        return 0;
    }
    else
    {
        return -1;  /* probably no event */
    }
}
```

where **event** is a pointer to a structure holding the result of the operation. If the read operation succeeds, 0 is returned, otherwise −1 is returned, indicating that the read failed. The following fragment of code illustrates access to the digital and analogue inputs, for the first two inputs.

```
void IOLib_UpdateIO(unsigned char DigitalOutputA, unsigned char DigitalOutputB)
{
  int e0, e1, e2;

  do
  {
    e0 = read_event(js0, &event);
    if (e0 == 0)
    {
```

```
        switch (event.type)
        {
          case JS_EVENT_BUTTON:
            switch (event.number)  /* joystick buttons/switches */
            {
              case 0:              /* A7 stick rear trigger (auto-pilot disconnect) */
                IOLib_DigitalDataA = (event.value) ? IOLib_DigitalDataA | BIT7 :
                                                     IOLib_DigitalDataA & ~BIT7;
              break;
              case 1:              /* A6 stick red button */
                IOLib_DigitalDataA = (event.value) ? IOLib_DigitalDataA | BIT6 :
                                                     IOLib_DigitalDataA & ~BIT6;
              break;
.
.
.
            }
            break;

          case JS_EVENT_AXIS:
            switch (event.number)     /* joystick analogue inputs */
            {
              case 0:                 /* P01 aileron */
                IOLib_AnalogueData[0] = convert(event.value);
                break;
              case 1:                 /* P02 elevator */
                IOLib_AnalogueData[1] = convert(event.value);
                break;
.
.
.
            }
            break;
```

This code is repeated for the USB ports **JS1** and **JS2**, where the function **convert** is used to acquire analogue data, converting the inputs to normalised (±1.0) values for use in the simulation. The simplicity of the interface facilitates quick adaptation of USB devices, while the overhead of reading inputs is of the order of a few microseconds.

References

Ahmad, M., Hussain, Z. L., Shah, S. I. A., and Shams, T. A. (2021), Estimation of Stability Parameters for Wide Body Aircraft Using Computational Techniques, *Applied Sciences*, Vol. 11, No. 2087.

Allen, L. (1993), *Evolution of Flight Simulation, AIAA Conf. Flight Simulation and Technologies*, AIAA-93-3545-CP, Monterey.

Allerton, D. J. (1996), Avionics, Systems Design and Simulation, *The Aeronautical Journal*, Vol. 100, No. 1000, pp. 439–48.

Allerton, D. J. (2000), Flight Simulation – Past, Present and Future, *The Aeronautical Journal*, Vol. 104, No. 1042, pp. 651–63.

Allerton, D. J. (2010), The Impact of Flight Simulation in Aerospace – A UK Perspective, *The Aeronautical Journal*, Vol. 106, No. 1065, pp. 607–18.

Anon. (1979), *The USAF Stability and Control Datcom*, User's Manual, AFFDL-TR-79-3032, Vol. 1, McDonnel Douglas, St Louis.

Anon. (1994), *Information Technology – Open Systems Interconnection – Basic Reference Model*, ISO/IEC 7498-1.

Anon. (2012), *I^2C-bus Specification and User Manual*, *Rev. 5*, available from https://www.nxp.com/docs/en/user-guide/UM10204.pdf (accessed 4 June 2022).

Anon. (2020), *Jane's All the World's Aircraft: In Service Yearbook,* 2020/2021 Edition, Jane's Group.

Burns, A. and Wellings, A. (2001), *Real-time Systems and Programming Languages*, Addison Wesley.

Cohen, L. L. (1955), *Report of the Court of Inquiry into the Accidents to the Comet Aircraft G-ALYP on 10 January 1954 and Comet G-ALYY on 8 April 1954*, Ministry of Transport and Civil Aviation (UK).

Donahoo, M. J. and Calvert, K. L. (2001), *TCP/IP Sockets in C*, Elsevier Sciences.

ESDU (2006), *Computer Program for Estimation of Aerofoil Characteristics at Subcritical Speeds: Lift-curve Slope, Zero-lift Incidence and Pitching Moment, Aerodynamic Centre and Drag Polar Minimum*, Item 06020, Engineering Sciences Data, London.

Hansen, P. B. (1973), *Operating System Principles*, Prentice-Hall.

Horowitz, P. and Hill, W. (1980), *The Art of Electronics*, Cambridge University Press.

IEEE Std 1003 (2017), *The Open Group Technical Standard Base Specifications*, Issue 7, IEEE.

Jacazio, G. and Balossini, G. (2005), Real-time Loading Actuator Control for an Advanced Aerospace Test Rig, *Journal of Systems and Control Engineering*, Vol. 1, No. 2, pp. 199–210.

Jackson, E. B. (1995), *Manual for a Workstation-based Generic Flight Simulation Program (Larcsim), Version 1.4*, NASA TM-110164.

Korn, G. A. and Korn, T. M. (1965), *Electronic Analog and Hybrid Computers*, McGraw-Hill.

Lewine, D. (1991), *POSIX Programmer's Guide*, O'Reilly and Associates Inc.

Link, E. A. (1930), *Combination Training Device for Student Aviators and Student Entertainment Apparatus*, US Patent Specification 1, Serial No. 825462.

Maekawa, M., Oldehoeft, A. E., and Oldehoeft, R. R. (1987), *Operating Systems - Advanced Concepts*, Benjamin Cummings.

McRuer, D. T. (1995), *Pilot-Induced Oscillations and Human Dynamic Behaviour*, NASA CR-4683.

Mitchell, C. G. B. (1973), *A Computer Programme to Predict the Stability and Control Characteristics of Subsonic Aircraft*, TR 73079, Royal Aircraft Establishment.

Moore, G. E. (1965), Cramming More Components onto Integrated Circuits, *Electronics*, Vol. 38, No. 8.

Roy, M. and Sliwa, S. M. (1983), *A Computer Program for Obtaining Configuration Plots from Digital Datcom Input Data*, NASA TM-84639.

Sloss, A. N., Symes, D., and Wright, C. (2004), *ARM System Developer's Guide*, Elsevier.

Smetana, F. O. (1984), *Computer-assisted Analysis of Aircraft Performance Stability and Control*, McGraw-Hill.

White, A. D. (1989), G-seat Heave Motion Cueing for Improved Handling in Helicopters, *AIAA Conf. Flight Simulation Technologies*, Boston.

Wilkinson, B. (1996), *Computer Architecture Design and Performance*, Prentice-Hall.