

AVIONIC SYSTEMS DESIGN

John R. Newport, Ph.D.



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 1994 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Library of Congress Card Number 94-11904

ISBN-13: 978-0-8493-2465-9 (hbk)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Publisher's Note

The publisher has gone to great lengths to ensure the quality of this reprint but points out that some imperfections in the original copies may be apparent.

Library of Congress Cataloging-in-Publication Data

Newport, John R.

Avionic systems design / John R. Newport

p. cm.

Includes bibliographical references and index.

ISBN 0-8493-2465-3

1. Avionics—Data processing. 2. Airplanes—Electronic equipment—Design and construction. 3. Microcomputers—Equipment and supplies. I. Title.

TL695.N48 1994

629.135—dc20

94-11904
CIP

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

DOI: 10.1201/9781315138008

TABLE OF CONTENTS

PART I: Introduction and Overview	1
PART II: Technology	13
Chapter 1	
Electronic Packaging	15
1.1 System Enclosures	15
1.1.1 Replaceable Unit Enclosures	16
1.1.2 Integrated Avionics Racks	16
1.2 Module Formats	17
1.3 Connectors	19
1.4 Wedge Locks, Other Mechanical Hardware	20
1.5 Design Issues	21
1.6 Packaging Significance	22
Chapter 2	
Microcircuit Technology	25
2.1 Memory	25
2.1.1 RAM	26
2.1.2 Battery Backed SRAM	29
2.1.3 EEPROM	29
2.1.4 EPROM Programming	30
2.1.5 Video RAM (VRAM)	31
2.2 Processors	31
2.2.1 Data Processors	31
2.2.2 Signal Processors	35
2.2.3 Graphics Processors	36
2.2.4 Controllers/Programmable Logic	37
2.3 Microcircuit Packaging	38
2.4 Multi-Chip Modules	39
2.5 Microcircuit Design	41
2.5.1 Design Languages	41
2.5.2 Behavioral Descriptions	42
2.5.3 Gate Level Descriptions	43
2.5.4 NetLists and Synthesis	43
2.5.5 Testing	44
Chapter 3	
Interconnect Technology	45
3.1 Parallel Interconnects	46
3.1.1 Pi-Bus	46
3.1.2 Futurebus+	48

3.1.3	VME	49
3.1.4	IEEE-488	51
3.1.5	Backplane Interconnect Electrical Design	51
3.2	Serial Interconnects	52
3.2.1	MIL-STD-1553	52
3.2.2	High Speed Data Bus	54
3.2.3	RS-232	55
3.2.4	TM Bus	55
3.2.5	ARINC	56
3.2.6	Microcircuit Test Channels	57
3.3	Sensor Interconnects	57
3.4	Transceivers	58
Chapter 4		
	Avionic Modules	61
4.1	Module Types	61
4.1.1	Data Processor Module	62
4.1.2	Digital Signal Processor (DSP) Modules	64
4.1.3	Graphics Processor Modules	65
4.1.4	I/O Modules	66
4.1.5	Memory Modules	67
4.1.6	Power Supply Modules	67
4.1.7	Mission Avionic Modules	68
4.2	Sample Module Descriptions	69
4.2.1	IBM DP	69
4.2.2	TI DP	70
4.2.3	TI quad C40	71
4.2.4	IBM Power Supply	72
4.3	Design Process	73
Chapter 5		
	Software	75
5.1	Software Requirements	76
5.2	Software Engineering Environment (SEE)	79
5.3	Software Management	83
5.4	Other Design Issues	85
5.4.1	Design Methods	85
5.4.2	Operating System Trends	87
5.4.3	Compiler Validation and Performance	90
5.4.4	Commercial Software Use	91
PART II Summary		95
PART II Exercises		97

PART III: Systems 101

Chapter 6

Communication Models	105
6.1 Message Passing	105
6.1.1 Bus Hardware Message Passing Model	106
6.1.2 Host Side Message Passing Model	107
6.1.3 Message Chains	108
6.1.4 Linked Chains	110
6.1.5 Message Passing Significance	111
6.2 Shared Memory	111
6.2.1 Shared Memory Coherency	112
6.2.2 Sample Local Memory Bus Operation	113
6.2.3 Shared Memory Message Scheduling	114
6.2.4 Shared Memory Performance	115
6.3 Networks	116
6.4 Communication Models Overview	119

Chapter 7

Architecture Integration Approaches	121
7.1 System Interconnects	122
7.1.1 LCMC	123
7.1.2 TCMP	125
7.1.3 Interconnect Selection Criteria	128
7.1.4 Interface Hierarchy	130
7.2 Processor-Memory Architectures	131
7.3 Module Architectures	135

Chapter 8

Design Examples	139
8.1 Retrofit Design Example	142
8.1.1 System Requirements	142
8.1.2 Definition of Candidates	144
8.1.3 Evaluation of Candidates	145
.1 Evaluation of Technology Upgrades	145
.2 F3 Evaluation	147
.3 MC Upgrade Evaluation	149
.4 Integrated Mission/Display Computer (IMDC)	150
.5 Other Impacts	152
8.1.4 Candidate Selection	154
8.1.5 Post Contract Award	155
8.1.6 Lessons Learned	156

8.2 New Tactical Aircraft System Example	157
8.2.1 System Requirements	157
8.2.2 Definition of Candidates	160
8.2.3 Evaluation of Candidates	162
.1 Federated Architecture	162
.2 Integrated Processor Evaluation	164
.3 Parallel Processor Evaluation	164
8.2.4 Candidate Selection	167
8.2.5 Lessons Learned	167
8.3 Special Mission Aircraft Design Example	168
8.3.1 System Requirements	169
8.3.2 Definition of Candidates	169
8.3.3 Evaluation of Candidates	170
.1 Message Passing Architecture	170
.2 SIMD Architecture	172
.3 Mesh Architecture	172
8.3.4 Selection of Candidates	174
8.3.5 Lessons Learned	175
PART III Summary	177
PART III Exercises	179
PART IV: Performance	181
Chapter 9	
Performance Analysis Methods	183
9.1 Benchmarks	184
9.1.1 Classical Host Side Performance Tests	185
.1 HOL Host Side Tests	185
.2 Assembly Language Host Side Tests	188
9.1.2 System Benchmarks	188
9.1.3 Benchmark Environment	190
9.2 Performance Measurement	192
9.2.1 Host Side Benchmark Results	192
9.2.2 Bus Side Benchmark Results	193
9.2.3 Results of System Benchmarks	194
9.3 Performance Estimation	197
9.3.1 Projections Based on Historical Data	198
9.3.2 Projections based on Code or Hardware Analysis	201
9.3.3 Queuing Theory	202
9.3.4 Bus Side Analysis	204
9.3.5 Mathematical Models for System Behavior	206
.1 Single Processor Model	206
.2 Multiple Processor Model	208
9.4 Lessons Learned	212

Chapter 10	
Analysis Examples	213
10.1 Interface and Functional Element Analysis	213
10.1.1 Interface Analysis	215
.1 Message Scheduling	216
.2 Message Latency	218
10.1.2 Functional Element Analysis	222
.1 Bus Side Analysis	223
.2 Host Side Analysis	226
10.2 Derived Component Requirements	228
10.2.1 System and Derived Requirements	229
10.2.2 Parametric Description of Requirements	230
10.2.3 Example	230
10.3 System Analysis Example	235
10.3.1 System Requirements	236
10.3.2 Description of Existing System	237
10.3.3 Description of Proposed Solution	237
10.3.4 Analysis of Proposed Solution	239
 PART IV Summary	 243
PART IV Exercises	245
 PART V: Systems Engineering Formal Methods	 249
Chapter 11	
Formal Methods	251
11.1 The Design Process	251
11.1.1 Process Maturity Model	251
11.1.2 Object Oriented Design	254
11.1.3 Statistical Process Control	257
11.1.4 Epilogue	258
11.2 Systems Engineering Deliverables	261
11.2.1 System Requirements Definition	262
11.2.2 Phases of a Development Program	263
11.2.3 Formal Reviews and Audits	266
11.2.4 Key Design Tasks	266
11.2.5 Technical Deliverables	273
11.2.6 Management Deliverables	274
11.3 Systems Engineering Technical Tasks	274
11.3.1 Requirements Analysis	275
11.3.2 Functional Analysis/Allocation	276
11.3.3 Synthesis	277
11.3.4 System Analysis and Control	279

Chapter 12	
Project Examples	281
12.1 New Tactical Aircraft Example	281
12.1.1 Review of Design Choices	281
12.1.2 SEMP Overview	282
12.1.3 Trade Studies	283
12.1.4 SEMS	284
12.1.5 Risk Analysis and TPM	286
12.1.6 Formal Reviews	287
12.1.7 Integration Teams	288
12.1.8 New Aircraft Overview	290
12.2 Tactical Aircraft Retrofit Example	290
12.2.1 Subsystem Master Schedule	291
12.2.2 System Specification	293
12.2.3 Risk Analysis	296
12.2.4 Formal Reviews	298
12.2.5 Retrofit Overview	299
12.3 Special Mission Aircraft Example	300
PART V Summary	303
PART V Exercises	305
PART VI: Trends and Summary	307
Technology Trends	307
Systems Design Trends	309
Policy and Formalism Trends	310
Unifying the Commercial and Military Computer Markets	312
Summary and Observations	316
Perspective	317
Abbreviations and Acronyms	319
Trademarks	325
Instructor's Notes	327
Index	329

AVIONIC MODULES

The avionic module is a hardware element within the avionics architecture which provides one or more architectural functions. These functions include data, signal, and graphics processing, mass storage such as global memory, and I/O interfaces to other clusters. In Air Force terminology, a module is a Line Replaceable Unit. In Navy terminology, an avionic hardware module is a Shop Replaceable Assembly. These names reflect the respective maintainability strategies of the two organizations. In this chapter, some specific vendor module types are described, which illustrate some standard functional element examples. An overview of the module design process is also be presented. Finally, technology trends are discussed, as well as a comparison to commercial computer systems.

Commercial computer board designs are discussed in the design examples of subsequent parts. The module types presented in the part are primarily SEM-E format designs, optimized for fighter/attack applications. As such, these examples offer insight into characterizations of system requirements for such aircraft.

4.1 MODULE TYPES

Prime contractor and avionics hardware vendors define avionic module types for a system as part of functional partitioning (see Chapter 13.3.2). Trade studies, requirements analysis, cost studies, software approach, and other factors are pivotal elements in the decision process. Relevant trade studies might include technology surveys, system interconnect selection, and performance analysis.

The PAVE PILLAR program, completed in 1987, was an Air Force study which addressed the impact of VHSIC and other emerging technologies on system architecture. The results of this study have been leveraged not just by the original target, the F-22, but also by many new and retrofit applications through the work of JIAWG and MASA. The common module approach which JIAWG matured evolved from the PAVE PILLAR core processing architecture. The PAVE PILLAR architecture is discussed in more detail in Chapter 7.

In the PAVE PILLAR core processing architecture, the various processor types discussed in Chapter 2 are mapped onto separate SEM-E modules. Each I/O and memory type also maps to a module type. Aircraft which implement this approach include the F-15E, EF-111, F-22, A-12, RAH-66, and others.

This design approach has been extended to mission equipment by the PAVE PACE program. In this part, a description of some of the core processing modules will be presented.

The PAVE PILLAR approach is normally implemented as a common module family unique to each program. However, the SEM-E modules designs can be quite complex, often emerging as multiple layer interconnect boards (10–12 layers), double sided. One side contains common elements, which means that each member of the module family has the same components and layout for this side. Typically, common side components include SRAM, backplane interconnect interface logic, and maintenance processor. The unique side contains the components necessary for the function which the module is to provide. In the case of the various processor module types, the unique side would contain the processor and additional memory. For I/O modules, the unique side would contain the serial interconnect logic and transmitters/receivers.

PAVE PACE is the program which follows the successful completion of PAVE PILLAR. One important goal of PAVE PACE is to extend the common module approach to the mission avionics area. Mission avionics includes the high performance sensors and sensor processing front ends, so the core avionics common module set cannot provide all the functionality needed for mission avionics.

In subsequent parts, examples of the various module types are described. The types include data, signal, and graphics processors, global I/O, memory, and power supplies. Also, extension of these approaches to mission avionics modules is presented.

4.1.1 DATA PROCESSOR MODULE

The data processor module architecture must reflect the system architectural approach, as Chapter 7.3 describes. Some examples for both loosely coupled and tightly coupled systems are topics. The precise architectural definitions of these terms are presented in a later chapter that describes processor and memory architectures. However, loosely coupled designs usually present characteristics such as scheduled block message groups with regular, periodic updates, while tightly coupled operations involve small or single word, fast memory accesses.

By far the most common integration approach for tactical aircraft is the loosely coupled system, in which each module is treated as a single board computer. The interconnects between modules, therefore, are highly layered and function in many ways like a local area network. Of course, extensive fault tolerance capabilities must be added to these interfaces. The PAVE PILLAR architecture is defined in this manner.

Variations of the Air Force PAVE PILLAR architecture are widespread. As discussed previously, in this design approach, a SEM-E module contains components common to the entire module family. These include a MIL-STD-1750 16 bit processor used as an I/O controller, a diagnostic processor (often

the same MIL-STD-1750 processor), or a bus interface unit. In many designs, the I/O processor and the bus interface unit are capable of message chaining, a concept that is described in a later chapter on the message passing communication model. This approach is highly efficient in loosely coupled systems and involves simply setting a pointer at the beginning and end of the block of data to be transferred. The protocol words are initialized at system start-up. The processor running the application code, therefore, does not have to wait on the I/O to be completed, but can return to application processing until an I/O completion interrupt is received. The bus interface units are configured to provide dual 16 bit error detecting Pi-Bus (32 bit for F-22), dual Test and Maintenance bus, and discretes of various types. The application specific portion of the module contains another MIL-STD-1750 processor and user memory. These modules normally provide 512–1,024 KBs of local memory used by the application software.

The power of the loosely coupled approach is evidenced by upgrade programs now in development. In many cases, it is unnecessary to replace all the modules in the computer. Instead, it is possible to upgrade module types individually. This is impossible in some other architectural approaches because the hardware functionality and performance is spread among several modules, instead of the functional encapsulation of the PAVE PILLAR and JIAWG approaches.

The next generation of data processor technology involves replacement of the 16 bit MIL-STD-1750 PAVE PILLAR processor with a 32 bit processor. This is the approach taken on the F-16 upgrade, in which the TI MIL-STD-1750 data processor from the common module set has evolved to the R-3000 processor. The memory architecture for this module provides 64 KBs each of data and instruction cache.

A more integrated approach is taken for the F-22 modules. Hybrid circuits are developed to increase the level of functional integration and to reduce the number of pin outs routed within the module layers. For example, the data processor hybrid contains two Intel 80960MX[®] processors, a diagnostic processor, and 2 MBs of memory, contained in a 2 × 4 inches package which dissipates 25 watts. Other components provide additional memory and I/O, such as the Delco 32 bit Pi-Bus. Thus, a powerful single board computer can be developed within approximately one half the board space of the previous generation MIL-STD-1750 families. The major drawback of this type of module is the high power dissipation (more about this later).

The single board computer approach is prevalent in aircraft such as the F-15E, F-16 Upgrade, F/A-18 (various subsystems), F-22, F-111, and RAH-66. However, this approach is quite different from the architecture used in commercial workstations and personal computers. The reasons for this will be discussed in the systems design examples, but it is instructive to examine the impact on the data processor module design if a tightly coupled systems design approach is pursued.

A tightly coupled multiprocessor system communicates through real-time access to a shared memory. Therefore, the primary difference in the data processor module design will be in the processor module memory architecture and I/O system.

The design of the memory architecture for a tightly coupled system must address the issue of data consistency. A common method to solve this problem is dedicated hardware that detects and signals potential data consistency problems due to multiple copies of the same data (or possibly code) present in the local memories of the individual processors. The most time critical of such circuitry is that connected to cache memories (or read/write buffer interfaces) due to the speed of such devices. When a memory segment must change to assure consistency, all processors with copies of this data segment stall, waiting for the update to complete. The later chapter on communication models describes this problem more fully in the discussion of shared memory.

Another important aspect of cache coherency is the bus interface unit. The memory bus used by the processor must be extended to all the multiple processors. Of course, if a single processor type is used in the system, the addressing and data transfer are much simpler than if many types of processors are used.

Although the tightly coupled approach does have avionics applications, the loosely coupled approach has benefits for tactical aircraft. The next part provides more insight into such benefits. For this reason, though, the PAVE PILLAR and JIAWG architectural concepts will be used on future designs. Upgrades to existing designs, such as the product descriptions provided at the end of this chapter, will likely form the building blocks for such systems.

4.1.2 DIGITAL SIGNAL PROCESSOR (DSP) MODULES

A DSP module is often architecturally similar to a DP module, except, obviously, for the processor type. However, in recent years, parallel processing technologies have greatly influenced DSP module architectures. Therefore, the generations tend to be characterized as single DSP modules, multiple, loosely coupled DSP modules, and parallel processors.

As just mentioned, a single DSP module is architecturally similar to the DP module. This module type would typically provide the DSP component, approximately 1,024 KB of SRAM, 128 KB of EEPROM for start-up and diagnostic code, a backplane interconnect I/O coprocessor (e.g., Pi-Bus, VME) and a maintenance processor.

A common processor selection for such modules is the TI TMS320C30 family, or the AT&T DSP32. However, there are custom design for specific applications, such as radar or acoustic processing, with much higher performance for specific functions. In addition, module prototypes are becoming available with newer processors which offer higher performance.

A loosely coupled, multiprocessor DSP extends this design by placing multiple DSPs and additional memory on a SEM-E module. Each DSP would

have a segment of private memory and the aggregate would also have global (shared) memory. This partitioning is normally programmable using the compilation system and operating system facilities. A typical design might have 4 DSPs with 4,096 KB SRAM. The other components listed for the single DSP module would also be retained.

In the realm of pure parallel processors, two types exist. The first is a Single Instruction, Multiple Data (SIMD) machine in which one assembly language instruction is executed simultaneously by a bank of identical processors. However, the data pipelines to these processors are unique. One mechanization of this approach is a common global code segment of memory, with a common instruction pointer for each processor. Such a design can be quite efficient with memory usage. This type of arrangement might be used for sensor processing.

Another type (less common) is the Multiple Instruction, Multiple Data (MIMD). A MIMD machine provides a bank of processors, potentially of different types, whose results are cycled to the other processors each clock cycle. Such machines can be quite powerful, but the software may be enormously complex. At the microcircuit level, this is equivalent to the super scalar approach of multiple logic unit types within the processor.

These design concepts are extensible to the system level. As such, the advantages and disadvantages of loose versus tight coupling, parallel versus vector or scalar processing, and software complexity will be discussed more fully in the next part. Also, descriptions of vendor DSP module designs are presented later in this chapter.

4.1.3 GRAPHICS PROCESSOR MODULES

The Graphics Processor Module architecture differs from the DP module design in two important ways. The first is that the module contains a graphics processor instead of a general purpose data processor. The second difference is the presence of video RAM instead of SRAM. These distinctions were discussed in Chapter 2.

The graphics processors described in the last chapter appear in avionic module designs. However, the most common Higher Order Language (HOL) used for the graphics modules is C, unlike data processor modules, which are programmed in Ada. Therefore, the new, high performance graphics and signal processors will cause C to once again emerge as a HOL candidate for avionics. The emergence of C++, which supports object oriented programming and other software formalisms, may help to ease this transition.

VRAM was also described in the last part. The distinctions between VRAM and SRAM is in the memory addressing mechanisms. However, both memory types are identical to commercially used components, except for packaging. Therefore, as mentioned in the last part, development of such memory devices can be leveraged from the commercial computer industry. In the same manner, a well conceived graphics or signal processor module design can simplify use of commercial software libraries.

As the last chapter explains, the TI SMJ34020 and Intel 80860™ dominate the avionics and prototype graphics system market. However, there are other options as well.

Some systems simply use a high performance data processor module for both graphics control and image processing. In general, video processors offer more flexibility and performance over a range of applications. However, if a data processor can meet the graphics requirements of the system baseline, then this approach provides a more common hardware and software environment, which can have a significant cost impact. As usual, careful requirements analysis and technology evaluations must be the basis of such a decision.

4.1.4 I/O MODULES

I/O modules serve as gateways to connect clusters of processor and memory modules together. The I/O module reformats signals to or from the backplane interconnect into a system serial interconnect. As Chapter 3 discusses, this function is necessary because high speed, parallel, electrical buses cannot be extended over large distance.

The most common I/O module provides a MIL-STD-1553 interface. A typical design would provide the MIL-STD-1553 receiver/transmitter and logic unit, 512 KB of SRAM, 128 KB of EEPROM, a backplane bus interface unit, and a maintenance processor. This module would be used to download software prior to system start-up, or to communicate with other integrated racks during system operation.

Another I/O module type provides the HSDB interface. The HSDB is described in the previous chapter. Architecturally, this module is similar to the MIL-STD-1553 module, except for the different receiver/transmitters and bus logic.

Discretes modules are also very common. Aircraft discretes are usually not Transistor-Transistor Logic (TTL), so some type of conversion is needed so that digital logic can process the information which such discretes present. An input resistor circuit is often set up so that the aircraft discrete can provide the proper current and voltage to turn on a diode. An optical transceiver is then placed on the module to optically couple the aircraft discretes. The output of the optical transceiver is designed to produce TTL signals compatible with computer digital logic. A discretes module will normally connect each discrete to a dedicated bit in a register, so that 16 discretes are read as a word onto the backplane interconnect. Other interfaces, such as RS-232, are sometimes also placed onto the discretes module.

Analog modules are also often needed for a systems design, such as those described in Chapter 9. The distinction between analog and digital modules for system interfaces is somewhat hazy, though, because aircraft discrete signals do not use conventional TTL trigger levels. Finally, there are module designs that provide multiple digital and analog interfaces, such as the TI Multiple Purpose Interface module.

The definition of I/O module types is an important step in the system architecture definition. These modules provide the system level communication among processing clusters. However, a proliferation of interface types can increase the power dissipation, weight, installed space, and cost of the entire system. In this area, as in all others, the systems designer must not let technology drive interconnect selections. Instead, a reasoned analysis of system requirements must be the basis for interface selection.

4.1.5 MEMORY MODULES

Two different memory module types are found predominantly in processing systems, including core avionics. These are bulk memory and extended memory.

The bulk memory module provides system mass storage for software loads and maintenance data. Some bulk memory modules provide on board processing to permit a file structure type access. These modules can usually act as both masters and slaves on the backplane interconnect. Another approach is to simply place the memory, slave only bus interface, and maintenance processor on the module. This means that the requesting module must provide memory management, including file structures, as needed. However, this approach is very efficient from a hardware standpoint since it maximizes the amount of memory on the module. The storage medium of such modules is often EEPROM, although battery backed SRAM is another option using the aircraft battery.

The Extended Memory Module (EMM) is used to provide additional RAM, typically to a DP module. The interface between the EMM and DP(s) is a high performance memory bus, such as that used by the data processor to communicate with its local memory. EMMs contain only SRAM, no other processors or bus interfaces are provided. With the coming of multi-chip hybrids and very dense SRAM, most designers are able to get sufficient memory on the DP modules. Therefore, EMM may not be necessary in future systems designs.

The F-22 uses a Global Bulk Memory. Despite the somewhat confusing name, this module serves as a multiported, large SRAM segment for connecting loosely coupled DP and DSP modules to sensor interfaces. This module provides rapid interconnects to the sensor systems as a shared memory element. More will be presented on this architecture in a later part.

4.1.6 POWER SUPPLY MODULES

Power Supply Modules provide the regulated voltage and current source needed to run the various core processing modules. The PSM accepts aircraft power, which is often either 28 volts DC, 115 volts AC (three phase), or 270 volts DC. The power supply must filter and rectify this input to isolate the digital modules from the electrical noise produced by the aircraft generators. In addition, a voltage conversion must be performed. Typical output voltages from the PSM to the other digital modules are ± 5 , although 15 and 28 volts are also used. To achieve low power dissipation, some digital systems may use 3.3 volts in the future.

Power supply outputs are rated in watts, typically around 250 for a current design. This means that if only 5 volts were delivered, that such a module could deliver 50 amps of current. The electrical needs of such a system can be gauged by noting that typical signal line currents are 100 mA (0.1 Amperes), so a module with a 306 pin connector could theoretically consume 30.6 amps. With higher pin counts, the power consumption may be worse.

Power supplies are also rated in terms of efficiency. This is a measure of how effectively the input power is converted. Typical efficiencies are around 85%. The loss is mostly dissipated as heat.

Core processing subsystems require multiple power supply modules not only due to current draw, but also for better system fault tolerance. However, this approach requires that some control mechanism be established so that the output of individual power supplies can be monitored and controlled. Discrete analog signal lines are used to control and communicate with these modules. For example, a current sharing line for each power supply provides an interface to a feedback loop so that each supply can adjust to system power demands. Typical PSM output discretes are power normal and power fail imminent, which are used by digital modules to assure enough time to save data and begin an orderly shutdown in the event of a power problem.

4.1.7 MISSION AVIONIC MODULES

Although the technologies described in this part are focused towards core processing, many of the same standardization trends evidenced in the design of this equipment are emerging in mission avionics. This trend is motivated by the observation that the sensor electronics are the source for the bulk of tactical aircraft avionics cost.

Mission avionic modules support the specialized I/O requirements and analog nature of some mission specific equipment. This type of electronics has largely been point designed for each application, although standardization is emerging for such equipment. The two ends of the mission equipment interface spectrum are low speed, mechanical sensors, and high speed, radio frequency communication devices.

One example of a low speed sensor is the accelerometer of older inertial navigation or flight control systems, which is mechanized as a pendulum whose displacement indicates the acceleration in the direction in which it can swing. Another example is the mechanical rate gyro, which is, in essence, a spinning flywheel which uses angular momentum conservation to measure torques applied to it. The most common communication interface for such sensors is the synchro-resolver, which provides an analog displacement measurement value. The sensors can be implemented so that the displacement is either linear or angular. Many companies make interface units for this function, such as the DDC RDC-19200 which offers up to 16 bit resolution in programmable sample rates.

Mission avionic modules for low latency, radio frequency applications include specialized receiver/transmitters, preprocessors, Analog to Digital

Converters (ADC), and fast analog (RF) memory. These elements are used for functions such as communication, radio navigation (Global Positioning System, Tactical Area Navigation), and identification (altitude encoding transponder). The PAVE PACE program contractors studied standardizing these functions in much the same way as standardization has proceeded in core processing. This will include standard packaging (SEM-E).

High performance, new technology ADC will revolutionize mission avionics. Much of the existing analog mission equipment will be pushed out of newer systems. New miniaturized power supplies and power converters will simplify packaging of these functions. Finally, the preprocessors will be miniaturized and achieve higher performance, which will further simplify packaging and increase functional density.

The low speed sensors are gradually being replaced by fiber optics and other technologies. However, the ADC/DAC will continue to be important for new designs.

4.2 SAMPLE MODULE DESCRIPTIONS

This discussion provides some examples of data processor, signal processor, and power supply modules.

4.2.1 IBM DP

The IBM Common Modules are a SEM-E form factor product originally designed for the F-15E and A-12. The processor is an IBM designed MIL-STD-1750 16 bit processor which is manufactured by Intel using its CHMOS process. The primary interface is provided by an Intel Pi-Bus controller, configured to switch between two identical 16 bit, error detecting buses. Other interfaces include TM bus, IEEE-488 console interface, various dedicated and programmable discretes, and power and grounds. An IBM custom designed Diagnostic and Maintenance Processor (DMP) performs on line diagnostics. The Instruction History gate array is provided to store information primarily used for software debug. 1,024 KB of SRAM and 32 KB of EEPROM, both parity protected, are provided. The SRAM is wired so that it can be battery backed. On the module memory bus provides a full 32 bit data path, with 23 address lines.

The DP module is shown in Figure 4-1. The picture shows the B side of the module, which is common for all of the IBM digital modules. This includes the MIL-STD-1750 processor (at the top, near the extractors), the Pi-Bus controller (center of the module, at the bottom), the DMP (to the right of the Pi-Bus controller), and the Instruction History gate array (the other large logic unit). The Pi-Bus transceivers are along the bottom, near the connector. The extractor levers are at the top and the wedge lock hardware, which is hidden behind the side rails, fasten the module in place.

This module weighs 1.2 pounds and generates 21 watts. The calculated Mean Time Between Failures (MTBF) is 17,380 hours. This module is designed for convection air cooling.

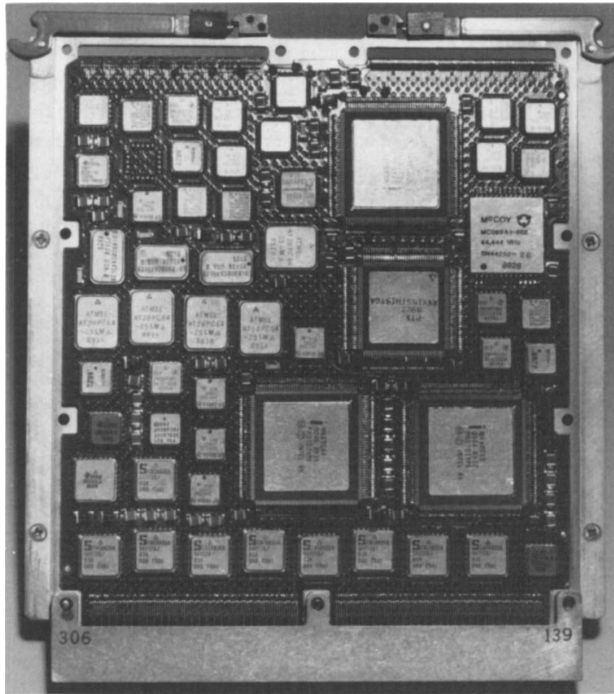


FIGURE 4-1. IBM DP module. (Courtesy of IBM.)

The TeleSoft MIL-STD-1750 Ada compilation system has been ported to this target. IBM developed this module for the A-12 and F-15E programs. It is also used on the EF-111, and is a candidate for others.

4.2.2 TI DP

The TI DP module shown in Figure 4-2 is a direct implementation of the PAVE PILLAR module architecture. Two identical TI MIL-STD-1750 16 bit processors are provided. One serves as the master CPU, and the second normally functions as an I/O processor, although this arrangement can be programmed using operating system procedures. Dual TI Pi-Bus controllers provide complete redundancy and independent bus A and bus B operation. 1,024 KB of SRAM and 64 KB of EEPROM are provided.

This module weighs 1.25 pounds, dissipates 18 watts, and has a calculated reliability in excess of 30,000 hours. This module is designed for convection air cooling.

The Tartan Ada system and TI Network Operating System ports are available for this target. TI developed this module for use on the YF-22A program.

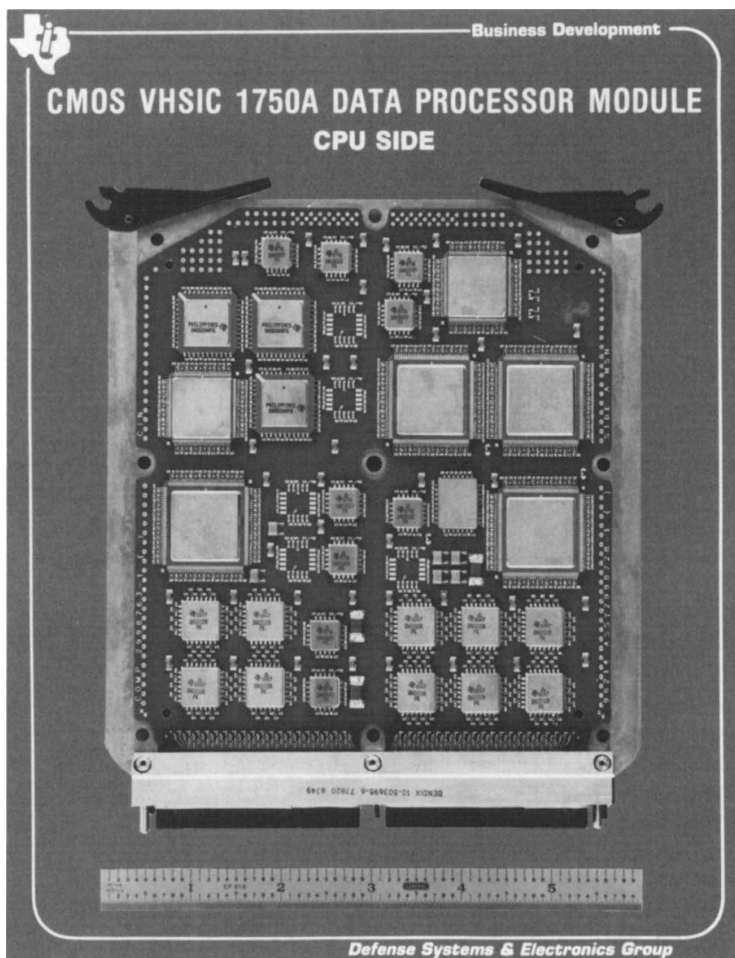


FIGURE 4-2. TI DP module. (Courtesy of Texas Instruments.)

4.2.3 TI QUAD C40

The TI quad-DSP module shown in Figure 4-3 provides four TMS320C40 digital signal processors. 5.5 MB of memory total is provided, with each C40 allocated 1 MB of dedicated SRAM, 1 MB global SRAM, and 512 KB of EEPROM. The module interfaces are VME as the backplane interconnect, VME System Bus (VSB) is the mass transfer bus, and RS232 as the console interface. Finally, the processor bus is also available at the card edge so that multiple C40 modules can be interconnected as a parallel processing machine.

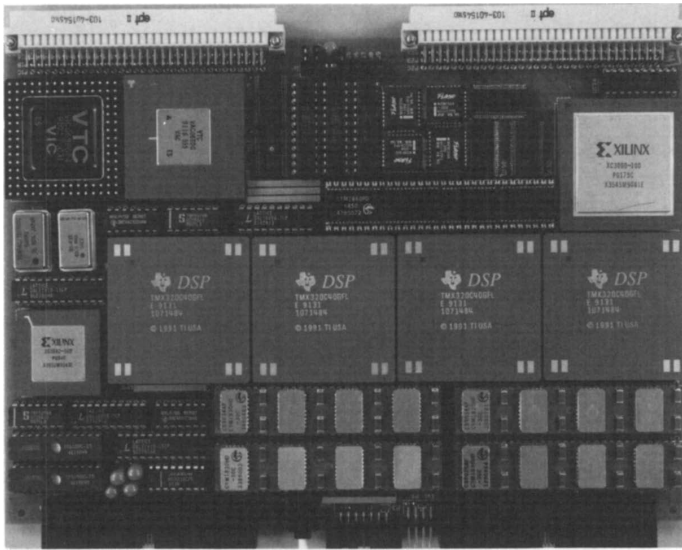


FIGURE 4-3. TI quad C40 module. (Courtesy of Texas Instruments.)

The package size is the VME standard 6U. The Tartan Ada and C compilation systems are available for C40 targets.

A dual C30 module is available from TI in a SEM-E format. This module provides 128 KB of SRAM and 8 KB EEPROM. The primary interface is the Small Computer System Interface (SCSI). This module was designed as an independent research effort and is based on the YF-22A common module design.

Both of these modules are convection air cooled.

4.2.4 IBM POWER SUPPLY

The IBM PS module for its Common Module family accepts 270 volts DC input and can produce 44 amperes at 5 volt output, at an 80% efficiency. The PS modules provide signal lines so that up to five modules can be used to provide current sharing. The module format is SEM-E, dissipates 55 watts, weighs 1.5 pounds, and has a calculated MTBF of 50,450 hours.

This module is shown in Figure 4-4. The module hardware, such as the extractor levers and connector, is identical to the hardware for the digital modules. However, the picture does provide some feeling for the complexity of the design. A power supply is an analog module. The placement of discrete components, such as the rather large capacitors along the lower right side, can have an important impact on both electrical and mechanical properties. The electrical properties can be altered by, among other factors, the length of the signal routings. The mechanical properties can be altered by mounting

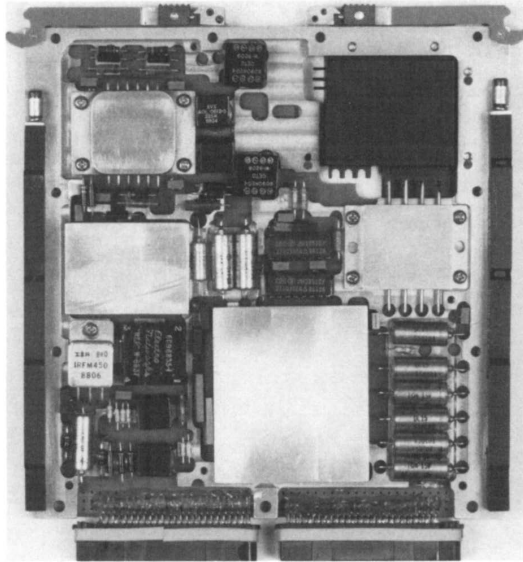


FIGURE 4-4. IBM PS module. (Courtesy of IBM.)

relatively massive devices near the rails, so that the lever arm associated with vibration is minimized. The devices that tend to generate more heat are also often placed near the rails for better dissipation.

4.3 DESIGN PROCESS

The design of avionic modules, similar to software and hardware components, has become highly automated. After completion of microcircuit designs, the modules are simulated by connecting together the microcircuit simulations. This allows detailed estimation of module electrical characteristics, such as impedance, capacitance, and transmission line effects versus clock speed and temperature.

An Automatic Place and Route (APR) software program provides an optimal, rapid solution for the module wire layouts and placement of microcircuits. This program treats the problem as a three dimensional grid through which interconnects must be routed. The resulting data files can be used directly by numerically controlled machines to produce the multiple layer circuit boards.

As with most complex systems, it is important to find problems early in the process before costly redesigns are needed. These problems may be design imperfections or poorly specified requirements. Extensive simulation and hardware prototyping is used to reduce risk in this area. This approach is discussed in the part on the systems engineering process related tasks.

ANALYSIS EXAMPLES

It may be intuitively clear from previous discussions that many different types of analysis are needed to develop a design baseline. One example is analysis of the communication within the system. The communication analysis is normally split into two segments, the system serial bus and the parallel module interconnect. Another type of analysis deal with the performance and functionality of a computer module, or functional element. A final example is the requirements analysis associated with an entire aircraft system. This chapter presents examples of such analysis.

10.1 INTERFACE AND FUNCTIONAL ELEMENT ANALYSIS

Many of the benchmarks in the last chapter deal with interaction of the processor and memory structures only. The system tests measure the relationship of processor performance to I/O activity. To complete the spectrum, another performance analysis realm is the interface hardware. This hardware analysis includes the behavior and speed of dedicated I/O memory and bus interface logic, as well as limitations imposed by the electrical characteristics of signal lines. This analysis also includes the interaction of computer functional elements within a communication network.

This chapter deals with the use of performance results, such as those described in the last chapter, to test primary requirements of architectures, which are interfaces, and functional elements.

The Interface Requirements Specification (IRS) for the Computer Software Configuration Items (CSCIs) specifies the message traffic and timing for each software unit. A CSCI is an application program. The functional and performance requirements for the CSCI are provided by the Software Requirements Specification (SRS). The systems engineers must produce the requirements for the IRS and SRS from those of the system specification. Historical examples of specifications are a strong influence in this process.

In the subsequent discussion, the focus is on the external hardware communication channels, not internal communication within a CSCI, which often executes on a single data processor module. These channels include the backplane bus and system buses.

Another important document is the Interface Control Document (ICD). This contains detailed, bit level word formats for each data word. For most systems, an ICD is developed for each system interface, for each computer subsystem.

Two important system features, message scheduling and hardware latency, are the primary outputs of interface analysis. Message scheduling relates the update rate for the Ada task performing I/O, to the time needed to actually execute this function with other tasks running. Hardware latency refers to the amount of time needed for a data word to be transferred from the originating software to the receiving software. However, the software and systems designers must define a message mix before one can obtain results from these two features.

A sample message mix is shown in Table 10-1. This represents the communication of a navigation function with the remainder of the system. Control information passes into the calculations for sensor updates. This includes fly over waypoints, radio navigation information, or relative navigation with data links. The control information also includes change of modes that might occur with a reset or off-line diagnostics. The data writes are the navigation solution outputs.

The following chapters use this message mix as the basis of numerous examples involving message scheduling and latency. These discussions are followed by a more general description of hardware interface and software operation. This revisits many of the architectural issues which provide the hardware technology basis for these discussions.

Functional analysis refers to the careful examination of the hardware and software provided to support some avionic functional requirement. Avionic functions include both core processing functions, which tend to be similar across platform types, and mission functions, which are unique to the platform type. Core functions include navigation, communication, identification, and flight control systems.

The purpose of functional analysis is to generate detailed hardware and software requirements for each avionic function, then to map these requirements onto a systems design baseline. Later in the development cycle, the analysis involves providing estimates that verify that the design can meet the requirements.

In highly integrated architectures, the distinction between an avionics function and one or more hardware modules is lost to the miniaturization inherent in the electronic packaging. In such systems, a hardware module, or small group, is called a *functional element*. The various buses through which the modules communicate are called *functional element interconnects*. Although this terminology can be somewhat confusing, it is commonly used in the design industry.

The confusion in terminology is most prevalent in derivations of a list of module hardware functions. These include data processors, signal processors, graphics processors, various I/O types, shared memory, and bulk memory. Once again, it is common to speak of a data processor functional element, although data processing is not a high level system function.

This chapter focuses on three general classes of functional elements. These are processing nodes, I/O nodes, and memory (storage) nodes. Previous discussions provide examples of each functional element type and its use. This

TABLE 10-1
The Navigation Message Mix

Message	Word count++	Read/write+
1	4	Write
2	2	Write
3	14	Write
4	28	Write
5	14	Read
6	7	Write
7	4	Read
10	4	Write
15	4	Write

Notes: + Read or write from the perspective of the navigation application.

++ 16 bit words.

chapter provides more detail regarding host side and bus side analysis. The four topics of this chapter are message scheduling, message latency, bus side functional element analysis, and host side functional element analysis. The subsequent discussion provides examples of both requirements validation and performance estimation.

The performance of the interfaces and functional elements must be closely matched for efficient operation. Although the dividing line may not always be clear, the distinction between interface and functional element analysis is that *interface analysis is based on the communication models, but functional element analysis relates to the architecture models*. The final chapter of this part explores this relationship more fully. In subsequent discussions, a cluster refers to a small number of functional elements interconnected by a cluster bus. A cluster communicates with other clusters with a system bus. The system architecture type, federated or integrated, determines the size and complexity of clusters.

10.1.1 INTERFACE ANALYSIS

Interface analysis provides an estimate for the amount of time needed to pass a message from one node to another within a system. The interface may be the cluster bus, such as a parallel backplane channel, or a system bus, such as MIL-STD-1553. Many hardware and software factors which impact this timing. These include software overhead, memory architecture, processor speed, and other factors.

This discussion develops technical descriptions of two important systems design issues, message scheduling and message latency.

After this chapter provides examples of such calculations, Chapter 10.4 completes the picture by describing the relationship of these parameters to system requirements.

10.1.1.1 Message Scheduling

The most comprehensive theory for message scheduling may be RMS, which matches closely the features of the message passing communication model. This is understandable since timings for the shared memory systems are set by hardware and timings for network systems are statistical. For this reason, the following discussion focuses on message passing.

Message scheduling refers to the rate at which the application software requests individual data transfers. In avionics applications, RMS theory is the most common method for both messages and software tasks. Once again, it is difficult to neatly separate message and application software scheduling since at least one of the application tasks performs I/O. RMS theory assumes that each software task is pre-emptible on the basis of task priority. The priority of each task is based on the requested update rate: the higher the rate, the higher the priority.

Since each task requires a certain amount of time to execute, it is possible to schedule tasks faster than the hardware can execute. If one of these tasks performs the I/O, a system deadlock might occur because the I/O task would never be serviced. A simple example can illustrate this point.

Suppose that a CSCI is composed of the following tasks:

- Executive and timing (EXEC)
- Navigation calculations (NAV)
- I/O for the navigation solutions (NAV I/O)
- A Kalman filter (KF)
- A Periodic Built-In Test (PBIT).

The times assigned for the execution of each task, the update rates, and priorities are shown in Table 10-2. The execution times are available from the SRS for the CSCI, but if it becomes necessary to calculate this value, two context switches and some idle time should be included (this discussion includes examples later). Table 10-2 also shows the priorities and update rates in Hertz (Hz) for each message. In this table, priority N is normal, which is machine and Ada system dependent.

The basic RMS theory¹ provides that the task schedule can complete within the allotted time if $\sum (C_i/T_i) \leq n (2^{1/n} - 1)$, in which C_i is the execution time, T_i is the scheduled period (inverse of rate), and n is the number of tasks, 5 in this case. The example of Table 10-2 is schedulable, because $\sum (C_i/T_i) = 0.2$, while the right hand side of the inequality is 0.74 for $n = 5$. If, on the other hand, all five tasks had been scheduled at 80 Hz, then $\sum (C_i/T_i) = 1.04$, which shows that the tasks are not guaranteed completion. Finally, if all the tasks had been scheduled at 40 Hz, then $\sum (C_i/T_i) = 0.52$, which is schedulable.

One note of caution, though. The RMS theory precedes widespread use of Ada tasking, so some generalization is necessary for an actual operating systems design. This example is intentionally simplified to show how RMS and tasking can be mixed.

TABLE 10-2
Software Tasks for the Example

Task	Execution time (ms)	Rate (Hertz)	Priority
EXEC	0.5	80	N + 1
NAV	1.5	40	N
NAV I/O	3.5	20	N - 1
KF	2.5	10	N - 2
PBIT	5.0	1	N - 3

Not only must the NAV I/O software task be scheduled with the operating system, its messages must be scheduled on the bus as well. System buses and some backplane buses offer hardware support for RMS scheduling. For such systems, message priorities relate to update rate, just as with the software. For this simple example, it probably does not make technical sense to schedule messages on the bus any faster than the calculations provide updates (20 Hz). Therefore, if the navigation message mix is the output of NAV I/O, then this message mix is scheduled on the bus at 20 Hz, with corresponding bus priority.

The bus hardware priority should not be used the same as software I/O task priority, although they are related. The bus priority provides a control input to the bus interface logic, while software priority is an input to the operating system software process scheduler. The bus message schedules may execute at a slower rate than the NAV I/O task, although the most sensible design is to schedule both at the same rate, if the system can achieve this. However, a good software design does not rely upon a fixed amount of time between task activation for I/O and the hardware bus transaction. This is known as a *critical timing loop* and can lead to *race conditions*, which is unpredictable sequencing, usually due to erratic timing for resources among I/O competitors.

Software scheduling involves parameters such as the amount of time needed to execute a task and scheduled rate. The bus scheduling involves similar parameters, except that execution time is replaced by theoretical bus throughput for the message mix.

If a 16 bit Pi-Bus is the interconnect, calculation of the theoretical update time is possible, as was done in Part II. the amount of time needed to transfer the message mix of Table 10-1 is:

Overhead/message	18 clocks
Waits/message	14 clocks
Total data cycles	81 clocks
Waits for data cycles	81 clocks

or Total = 9 · (18 + 14) + 81 + 81 clocks at 12.5 MHz, or 450 clock cycles at 80 ns, giving 36 µs total.

TABLE 10-3
Sample Pi-Bus Message Mix

Task	Execution time (μs)	Rate (Hertz)	Priority (logical bus priority)
NAV I/O	36	20	8
A1	800	40	9
A2	1050	40	9
A3	650	40	9
A4	10	80	10

Suppose that a computer system includes NAV I/O and the other messages shown in Table 10-3. When the bus time is multiplied by the update rates, the sum is 0.10152. This can be compared to the schedulability criterion for five processes of 0.74, as before with the software schedule. In fact, if all the message group schedules run at the same rate, the schedulability value (0.74349) divided by the total time (2.546 ms) yields a rate of 292 Hertz!

This example shows the impact of software overhead. The bus hardware is not 300 times faster than the host side memory and processor bus. In fact, from the perspective of the processing node, the Pi-Bus can be modeled as 80 ns memory. The difference in speeds is the amount of time needed to enter the appropriate software tasks, set or clear interrupts, switch contexts, and similar factors. Therefore, the theoretical bus throughput is not a clear indicator of the amount of time needed to actually transfer data from one memory system to another. This time is called *data latency*, and is the next topic.

10.1.1.2 Message Latency

Message latency is the amount of time needed to route a data packet from the source software task to the destination software task. The destination may be on a common interface with the source, or the connection may involve switching many different nodes. Routing in the message passing model is most tractable, so this model is the basis of the following discussion. Complex routings in shared memory systems must be avoided due to performance issues. In networks, complex routings are very likely, and queuing theory can be an appropriate theoretical basis.

In general, both hardware and software overhead contributes to overall latency. The I/O is most likely handled by a software task which runs at a priority determined by the system context. A message routing latency calculation must include the following for such a system:

- Source host side scheduling delay and task idle time
- Source host side software overhead to perform context switches, services interrupts, and perform the data transfer from user memory to I/O memory

- Bus hardware latency due to overhead and finite response time
- Destination host side scheduling delay and task idle time
- Destination host side software overhead.

For a message passing system, refer to Figure 6-1.

The *host side* refers to the data processor memory and processor system on which the software task runs. The *bus side* refers to the bus interface control logic, dedicated I/O memory, and other bus hardware such as transceivers and signal lines.

The host side software must schedule the data transfer, as the last topic describes. The operating system may not be able to service the request at once if other higher priority tasks are running or blocked. This situation will result in the I/O task being blocked by the higher priority process. In the tasking scenario of the last topic (Table 10-2), the NAV I/O must wait a maximum of 2.0 milliseconds for the EXEC and NAV to run. The wait will be smaller if either of these tasks suspends. For example, EXEC might be coded to simply update the timer, delay until NAV, NAV I/O, and KF signal completion, then run diagnostics and perform error recovery, as needed. There are other scheduling algorithms which can minimize waits, such as Fastest First,² but RMS scheduling is more common for avionics and is the basis of this discussion.

When the NAV I/O task begins to execute, some other task must be suspended since the data processor can only run one task at a time. The process state of the suspended task must be saved to memory so that it can be resumed later. After the current task is saved, the NAV I/O task process state must be loaded and started. This operation is called a *context switch*. In addition to the context switch, it may be necessary to signal the low level bus hardware by setting a service request interrupt, which causes a latency due to the time needed for the interrupt service routine. Finally, the data and protocol words must transfer from the application program memory to I/O memory, which the bus interface logic can access. This transfer occurs via the host side processor/memory bus. Also, if the NAV I/O task generates memory addresses which are not in cache memory, then a *cache miss* occurs, which must be handled with additional memory transfers. When the application program transfers from cache to I/O memory, a cache memory miss is likely, resulting in additional wait time. All of these factors (context switches, interrupts, memory access, and cache misses) contribute to host side software overhead.

The bus side latency refers to the speed which the interface control logic can perform the transfer. The previous topic provides examples for Pi-Bus and Futurebus+.

Finally, the destination task has host side latencies similar to those for the source task. The destination node interface logic must recognize and respond to the source task node, transfer the information from slave I/O memory to application memory, and signal the destination task.

In practical terms, it can be very difficult to provide accurate values for some of these parameters. The following numbers should be available from the computer hardware vendor or Ada vendor:

- Context switch time
- Interrupt latency
- Cache miss penalty
- Processor clock cycles needed for reads and writes.

However, the systems designer must calculate for the particular system context the scheduling delay and task idle time, as well as the hardware latency.

The previous chapter describes the parameters needed to make such estimates. These are the actual execution time needed for the hardware to perform the service, the scheduled update rate, and the priority.

Mathematical models of queues can be the basis of scheduling delay and idle time estimates. This approach may, or may not, involve statistical, discrete time simulation described in Part II. The arrival rates of requests to these queues are based upon the control flow of the tasks running at the time of the request. A model for the bus side system context is similar.

Once again, for the tasking example shown in Table 10-2, the sequence shown in Figure 10-1 would execute.³ In this example, the minor frame rate is 12.5 ms (80 Hz) and the minor frame for which all tasks execute is shown. The tasks need not be scheduled this way. Instead, processor loading can be averaged. However, this diagram represents the worst-case delay, which should always be the basis for such an estimate. In this diagram, the scheduling delay is 2.0 ms for the NAV I/O task. In a similar manner, Table 10-3 indicates that the worst-case delay for the bus transfer is 2.51 ms.

The analyst might derive a software overhead estimate from the vendor information as:

- Context switches (2 needed), 0.1 ms
- Interrupt latency (2 needed), 0.2 ms
- Cache miss penalty, 20%
- Memory transfer time (assume 40 ns memory for each read/write, 99 words, 2 clock cycles per transfer) 0.00792 ms

for a total of 0.30792 ms, or 0.3849 ms with cache penalty.

For this rather simple example, the source and bus delays are (the bus hardware latency is contained in the previous topic):

- Source host side scheduling delay = 2.000 ms
- Source host side software overhead = 0.385 ms
- Bus scheduling latency = 2.510
- Bus hardware latency = 0.036 ms,

TASK	EXECUTION TIME (ms)	SYSTEM TIME (ms)
(Start of Minor Frame)	—	0.0
EXEC	0.5	
(End of EXEC)		0.5
NAV	1.5	
(End of Nav)		2.0
NAV I/O	3.5	
(End of NAV I/O)		6.0
KF	2.5	
(End of KF)		8.5
PBIT	5.0	
(End of PBIT)		13.5

FIGURE 10-1. Sample execution timings.

or 4.931 ms for source task to destination interface logic. If the destination host side delays are comparable to those of the source (a task list must be provided for the destination), then the total message latency is 9.862 ms.

Many analysts (or marketing agents) often simply compute the amount of time needed to complete the data transfer cycles on the bus hardware. In addition, the average scheduling delay, $13.5/5 = 2.7$ ms, is sometimes used. This would produce a latency estimate of 2.736 ms, much too low. Customers of such analysts are often surprised that it is impossible to come close to the predicted performance. Worst-case timings should always be used in design analysis. This example is quite simple, but it illustrates two important concepts.

1. The overall message latency is much larger than the bus hardware transfer time;
2. The software and scheduling delays dominate the message latency.

One final aspect must be discussed. The NAV_I/O is scheduled at 50 ms intervals (20 Hz), but the latency is 9.862 ms. This represents a 20% utilization of this rate by this task, which is ordinarily adequate. However, the system requirements may dictate a lower utilization to preserve resources for further growth or time sequencing in the system. That is, I/O is normally scheduled at a latency low enough to assure a high probability of successful service. A designer can solve the latter problem another way.

Many algorithms must sequence messages correctly, as well as not miss any message arrivals, in order to produce correct outputs. It could happen, for example, that the destination misses one message group because it is servicing a high priority hardware event. In this case, the algorithm may be presented with the old data from the last minor frame. The new data from the next minor frame, therefore, may represent a discontinuity.

As an example, suppose the sensor inputs are linear, and the algorithm simply calculates and outputs the slope. The slope might be an output used for

a hardware control loop. A correct sequence, such as {0.0, 1.0, 2.0, 3.0,...} for sequential minor frames, produces the slope 1.0. If a minor frame is missed, though, the inputs become 0.0, {1.0, 2.0, 2.0, 4.0,...} Therefore, the first inputs provide the correct slope of 1.0, the next produces a slope of 0.0, and the last produces a slope of 2.0. If the slope is used as a control signal output, these changing values could cause oscillations or divergence, which is unacceptable.

The approach to this problem is *message time tagging*. In this method, a data word is added at the beginning of each message which is the real-time clock value on the host side when the message service request was entered on the message originating agent. This can be made much more precise for some systems. Many systems provide hardware which measures and provides a system time, which accounts for skew between nodes, clock skew and drift, and similar problems. However, for many algorithms, the time difference, not the absolute time, is needed.

An algorithm can be provided in software which monitors the message time tag and performs interpolations to minor frame boundaries. This is called *sensor deskewing*, regardless of the source of the data. In the slope calculation just described, the sensor deskewing algorithm would see that time stamps for two successive frames were identical. This increments a *minor frame miss counter* to be used for the overall time delay. The old slope would be used as the output for frames in which a minor frame miss occurs. At the next minor frame, when new data arrives, the difference in sensor data are divided by the counter: $(4.0 - 2.0)/\text{counter}$, in which counter is now 2.0, not the value of 1.0 for no misses. This produces the correct slope of 1.0. The algorithm then resets the miss counter.

There are many other applications of deskewing. Some algorithms produce a mathematical result which is at the middle of the minor frame, not the end. An example is a midpoint algorithm for differentiation. It may be critical to the accuracy of such algorithms that sensor values be interpolated to the time point of the calculation output.

10.1.2 FUNCTIONAL ELEMENT ANALYSIS

For the purpose of this discussion, a functional element is equivalent to a computer hardware module of a specific type (see the discussion in the introduction). Examples are data processor modules, specific I/O modules such as MIL-STD-1553, bulk memory modules, and others. The analysis associated with the functional elements includes performance analysis (especially throughputs and storage needs), interface analysis, and system factors such as reliability, maintainability, supportability, and other issues.

However, functional analysis does not repeat the detailed examination of these areas, but rather builds on the inputs from these specialty areas. The system level interaction of these topics is the perspective of this discussion.

For tractability, the discussion assumes a loosely coupled architectural approach. As the previous chapter suggests, the shared memory communication model is often the basis for a tightly coupled system, so performance issues

demand that the designer avoid complex interactions of buses and software in such systems. A loosely coupled system implemented as a network is statistical in nature. However, analysis similar to that which follows can determine many of the parameters that the analyst needs for the queuing theory models for networks.

For convenience of presentation, the functional element discussion consists of bus side and host side topics to mirror the interface analysis discussion. The relationship of system requirements to analytical results is the topic of Chapter 10.3.

10.1.2.1 Bus Side Analysis

The *bus side* of the functional element refers to the bus interface controller logic, reserved I/O memory, and cabling or signal lines which extend from the module. Analysis must be performed for this type of hardware which involves both system buses and backplane buses. Examples of both types are described later.

Chapter 3 and other previous discussions present a general description of bus side operation. This chapter probes more deeply into the system level factors that impact interface performance. These factors include:

- Architectural implementation
- Physical and electrical characteristics of a communications channel
- Electronic packaging
- Communication protocol modes.

An earlier discussion presents architectural topologies and implementation factors. It is inefficient to revisit this rather lengthy and complex discussion even though this presentation touches upon many of the same issues. Also, system analysis is the topic of Chapter 10.3, so this discussion presents only a preview of formal methods a designer needs to solve functional element problems. This discussion focuses on methods of behavioral modeling and analysis of operations on a single module type.

The topology by which a designer interconnects system nodes is as important as the message protocols. Each node and system communication channel must provide sufficient performance and functionality so as to avoid a system bottleneck. A very common example is the interconnect between a high performance backplane bus and a serial bus, such as MIL-STD-1553. More concretely, a 32 bit parallel bus might attain 15 MB/sec throughput, including overhead. If this bus were directly connected, such as through a shared memory interface, to a MIL-STD-1553, 1 MHz, serial bus (i.e., 0.125 MB/sec data clock rate, much less with software overhead), then the performance of the parallel bus would be wasted. One solution is to buffer the transfers by collecting MIL-STD-1553 inputs over time until enough data are accumulated to warrant a parallel bus transmission. During the interim, the parallel bus can service the local, cluster interconnect, as well as other system interfaces, such

as serial fiber optic buses. At the other extreme, another idea is to simply use one bus for all functional element and cluster interconnects. This approach is gaining popularity because of the emergence of massively parallel computer architectures.

The systems engineer must identify bottlenecks because they limit growth and deliverable performance. Additional system capacity can eliminate bottlenecks.

A related concern is the potential for a single point failure. The designer must fashion the system topology so that failure of any single node or link cannot render the entire system inoperable. This issue presents a serious design problem to implementation of large numbers of point to point links, such as might occur in a massively parallel system.

The systems engineer must identify single point failures because of their potentially catastrophic impact to system operation. Additional routing capacity can solve this problem. This capacity may simply consist of more channels, or it may require a different type of channel.

In addressing these issues, the systems engineer must rely upon realistic performance estimates. For example, performance calculations must not rely entirely on behavioral simulations of the communication protocol, since theoretical bus performance is impossible to achieve because of practical physical limitations. Specifically, there is a simple relationship between capacitance and attainable transfer speed. The two main factors which determine capacitance are, for many designs, either the transceiver logic or the number of modules in the cluster. Once a signal is past the transceiver, the laws of Physics, not protocol operation, determine maximum update rate.

Therefore, the transceiver design is an important, but overlooked, communications performance issue. Modern transceivers incorporate special logic arrays which minimize capacitance. This allows high performance buses to communicate more efficiently with high speed memory and processors. Currently, a safe upper limit for bus side designs is 40 ns response, or 25 MHz. Since advanced processors, and corresponding memory systems, operate much faster than this, the bus transactions must be buffered. The designer can accomplish this in at least two ways. First, dedicated I/O memory serves as a buffer. When the buffer fills up, an entire block transfers to the program memory from which the processor runs. Second, the bus interface control logic functions with minimal intervention by the host side processor. In message passing architectures, in particular, the host side processor usually initializes protocol words, message size, and a pointer to the start of the data block. During program execution, the application program updates the data block. When the application completes this, the processor simply tells the bus interface unit to send the block, which performs the transfer of the data to dedicated I/O memory and, ultimately, on to its destination elsewhere in the system. During the transfer, the processor can perform other tasks, which avoids processor wait cycles. When the transfer is complete, the bus interface unit signals the processor by setting an interrupt.

This message scenario is very common for tactical aircraft, but simply illustrates a more general rule. Recall that the three communication models of Chapter 6 are message passing, shared memory, and networks. The message mix of the application software must be the touchstone of selecting a communication model. Many of the bus side design issues stem from an inappropriate match between the communication model and the message mix.

Another physical design impact is the length of the bus, or, as some systems designers prefer to think of the problem, the number of modules contained in the cluster. Once again, two effects are primarily responsible for this design impact. The first is simply the end to end propagation delay of the bus. Good design practice dictates that no module transmit faster than this end to end time to avoid overlapping messages. In fact, two end to end times are really needed for complete signal isolation because of possible reflections from the bus termination circuit. The other factor is the capacitance provided by each module. The connector pins and traces on the module are obviously conductors, while the module structure is a dielectric. The separation of conductors by a dielectric produces a capacitance by each module which affects the entire bus system.

Electronic packaging is a closely related issue. The module size dictates the connector selection, which limits the achievable bus complexity. For example, if the logic needed for a particular bus type requires an entire side of any module, then this bus selection severely limits how much capability can be built into each functional element. One approach might be to limit module memory, although memory size is normally a crucially needed resource to match software requirements. In reality, there are no good alternatives for this dilemma. It may be more efficient to look for other bus types in such situations.

Another problem is signal routing. Each signal line requires ground pins so that sufficient electrical isolation occurs for good signal quality. A common design rule is about three signal pins per ground, or a reduction in available pin count by about 25%. Therefore, if the bus is very complex, uses a large address bus, or provides a very wide data path, then the bus may be impossible to implement in the electronic packaging approach chosen for the system.

The communication protocol modes also have an important consequence. Most buses operate in many different modes, but only a few modes are likely to be efficient for a given architecture. For example, MIL-STD-1553 hardware supports polling, but in message passing systems, this generates tremendous protocol overhead. Polling is really intended for systems in which MIL-STD-1553 is used as a point to point link, or with a very small number of terminals. Another example is the Pi-Bus suspend, which allows large messages to be broken up and transmitted in parts so that other nodes can gain access to the bus. The normal use of suspend, though, is not good design practice because of the protocol and software overhead needed to save the bus state, start and complete another communication process, and then resume the original message.

Many other system issues also play a role in bus selection and system implementation. Reliability is an example: the greater complexity in terms of

logic gate counts and signal line interconnects of the circuitry, the more likely the failure of the system. This again drives a systems designer to select simple buses. Maintainability is also an issue. The larger the number of pins in the connector, the larger is the force needed to insert or extract the module. The probability of computer damage by service technicians is linked directly to high insertion forces. Once again, simplicity is an important design consideration. Data security requirements may also limit the candidates that a designer can use.

Therefore, one system level issue is to identify any performance bottlenecks or single point failures. Also, physical characteristics, not protocol features, may be the limiting performance factor. However, dedicated I/O memory that serves as buffers can increase application software performance. The communication model on which the systems design rests must be appropriate for the message mix. Finally, communication channel complexity is related to system level reliability and maintainability.

This chapter has explored the architectural and packaging aspects which drive bus selection and interconnect choices. The *KISS rule* (Keep it Simple, Stupid!) is a general and important design consideration. This rule applies not just to the bus side, but also to the host side, as the next topic shows. The functional element analysis on the bus side includes not just the performance estimate from interface analysis, but also issues such as fault tolerance, reliability, and maintainability.

10.1.2.2 Host Side Analysis

Previous discussions provide general descriptions of host side analysis. This discussion explores host side analysis in terms of its system level impact. A detailed description of software execution is relevant to functional element analysis, including the execution time allocation of the I/O task. This causes some overlap with performance analysis, just as the previous topic overlaps interface analysis. The purpose of this discussion is to provide some design issues of system level host side analysis.

The host side refers to the processor on which the software executes, its application program memory, any other supporting hardware, and the host side memory bus. The processor may be a general purpose data processor, signal processor, graphics processor, or other special hardware (described later). The application program memory refers to the hardware from which user programs execute. Supporting hardware may be other special processor types, such as a diagnostic processor, or other controllers, timers, or supporting hardware. Finally, the host side memory bus is the communication channel that connects these hardware elements. The processor normally performs the memory management itself and therefore functions as the memory bus controller. This bus is typically a parallel bus, 32 bits (or more) wide, which operates synchronously at processor and memory clock speed.

The speed at which the processor can execute native instructions is an important performance parameter. A hardware vendor often provides clock

cycles needed per instruction. However, pipelines and multiple execution units can greatly speed up the execution rate. For modern, military, RISC data processors, this principle allows the average execution rate to be at or below one instruction per clock cycle.

Generally, the memory system includes local memory and cache memories. The cache memories must be *zero wait state*, which means that the cycle time (not just access time) for a read or write must be faster than one processor clock. If this is not true, then the processor stalls on each access and must insert wait states. However, fast memory is expensive, so while cache may be zero wait state, local memory is usually not. As a result, when a cache miss occurs, the application suffers a performance penalty because of the overhead associated with flushing cache to local memory, as well as the slower speed of local memory. These trade-offs underscore the importance of carefully selecting the processor-memory architecture best suited to the application. See Part III to revisit the candidate types.

Supporting hardware is unique to the application for the systems design. Examples are microcontrollers, memory controllers, or other logic types.

Finally, the memory bus is the primary connect between the processor and memory systems. Most hardware vendors provide detailed electrical and behavioral descriptions of the memory interface in the processor data sheets. This information allows the designer to calculate cache miss penalty, and related performance parameters.

This background, provides the basis for generating the software execution time requirements, such as those in Table 10-3. The following factors supplement the hardware information:

- Benchmark results from the Ada vendors
- Historical data
- Other analysis, such as simulation.

Refer to Chapter 9 for methods of performance analysis.

Although this presentation does not revisit the performance issues in detail, there are some additional estimating methods that can be useful. The following procedure can provide an initial estimate. First, estimate the number of lines of code in the application, which is normally available very early in the design. Next, find a benchmark with characteristics similar to the application. For example, if the application intensively uses floating point operations, Whetstone might be a good benchmark for this use. Next, determine for the benchmark code size and execution time a hardware system as close as possible to the new design. Also, use the same Ada compiler for the benchmark as will be used for the application, if possible. The code size (i.e., not program size, but code segment of the link map) corresponds very roughly to one instruction per word (not byte) of memory needed. The ratio of code size and execution time therefore gives an approximation to the average number of instructions per

second which the processor can execute. This result can be scaled the size of the application relative to the benchmark to produce an execution time.

Suppose a benchmark consists of 1,000 source lines, produces 23K words of code when compiled, and executes on a system in 100 μ s. For an application estimated to contain 15,000 source lines, the code expansion factor from the benchmark suggests that the application code size will be 345K words of code. Finally, the execution speed of the benchmark suggests that the application will require $100 \mu\text{s} \cdot (345/23) = 1.5 \text{ ms}$.

The analyst must compare these calculations to historical results available as a check. Experience shows, for example, that the relation between source lines of code and instructions varies considerably within a typical range of 0.4 to 1.6. In addition, select avionics code to use for a benchmark, if available. Finally, avoid extravagant extrapolation in sizes and speeds beyond factors of 10 to 15.

Once the new application software is available, the systems designer can refine these estimates considerably. The compiled code provides insight into performance, as the previous part describes. Therefore, the host side analysis is an iterative process.

The system requirements often include a reserve capacity, typically 50% for a new systems design. In the case just described, the reserve requirements require that 3.0 ms, not 1.5 ms, be set aside for this program. Reserve requirements are usually not included in schedulability analysis, though, since being too conservative can result in very costly hardware to meet the performance requirements.

The design issues that impact host side analysis include processor-memory architecture, software requirements, and performance reserve requirements. In addition, application specific requirements may be difficult to meet. Data security may be an example.

10.2 DERIVED COMPONENT REQUIREMENTS

The requirements process is one of the topics of the next part. However, a brief overview may be helpful. The customer, which is the Department of Defense for tactical aircraft, provides a weapon system specification to the aircraft prime contractor as part of the procurement package (i.e., contract). This massive package contains high level requirements for all aircraft equipment, including engines, airframe, ejection seats, auxiliary power units, landing gear system, and so on. Avionics often merits a single volume of a total of twenty or more. Without the incentive provided by numerous horror stories, program managers might miss the documents containing avionics requirements entirely.

The avionics volume matches top level functions with aircraft missions and mission phases. The prime contractor must develop these requirements into *detail specifications* for separate computer components. The detail specifications

are the basis of prime contractor awards to subcontractors for equipment development and purchase. The subcontractors must relate the requirements in the detail specifications to aspects of components for the design. Therefore, this lowest tier of documents contains *derived component requirements*.

All derived requirements must be traceable back to the weapon system requirements. If this process performs properly, any requirement which cannot be so traced does not contribute in an important manner to the aircraft mission, and the systems engineer should delete it.

This chapter presents some sample system and derived requirements. The presentation then relates the parameters described in the interface and functional element analysis to these system requirements. Finally, the text includes some examples.

A more detailed description of cluster and system requirements analysis and test are provided in later parts.

10.2.1 SYSTEM AND DERIVED REQUIREMENTS

One of the main system requirements that flows down into module design trade-offs is reserves. Normally, the system specification requires that a design offer a 50% reserve in order to assure room for future growth, as well as requirements changes during the design phase. Reserve requirements include the three areas of processor throughput, I/O channel throughput, and memory size versus code size.

In addition, the system requirements normally state that the design must meet the reserve capacity under worst-case loading, not average loading. This can become a significant issue during requirements analysis because there are often large peaks and valleys in resource utilization over time. In addition, some statistical simulations produce time averaged results, not instantaneous utilization (see Chapter 9). Therefore, the results of such simulations are not appropriate for requirements validation.

In the early stages of development, high level software requirements are the basis of allocations to hardware. Often, this corresponds to simply a list of software functions which the system must perform and an estimate of source lines of code. The systems engineer can combine these estimates with historical avionics functional performance data to produce a list of processor, I/O, and memory requirements for each major function. The team continues the design process by mapping the major functions onto hardware on the basis of performance estimates from benchmark results. The next topic describes this process in more detail.

One system level design goal should be to minimize the number of hardware modules and module types in order to control product price. These factors drive the ultimate bus, processor, and memory component selections. Earlier parts provide examples of such design issues. Once again, reserve estimates are an important factor in such selections because the common module designs must be based on the worst-case loadings for the entire system.

10.2.2 PARAMETRIC DESCRIPTION OF REQUIREMENTS

In this topic, the relationship of some of the performance parameters and the functional element requirements is examined. Many of the issues previously mentioned, such as architectural implementation, electronic packaging, and other system issues, are discussed at length earlier in this volume, and are not further explored in this part.

Once again, formal systems engineering methods are the topic of Chapter 11. However, one aspect of a successful design process is the capability to track Technical Performance Measures (TPMs), as described in MIL-STD-499B. A list of TPMs is a formal method, suitable for management use in design audits. The parameters this part describes are examples of suitable candidates as TPMs.

The first parameter is processor throughput. Processor throughput estimates can be based on at least the three key elements of compiled code analysis, simulation, and historical data. Refer to Chapter 9 for more details on performance analysis.

If an Ada compiler is available for the target processor, an analyst often bases an initial estimate on code analysis of standard benchmarks. That is, the number of clocks per low level instruction, provided by the manufacturer, can be the basis of a performance estimate. This estimate usually requires refinement to reflect the performance gains attributable to pipelines or multiple execution units. Once again, the hardware vendor can supply this information.

A software execution scenario is another necessity for throughput estimation. This must include a timeline of features such as tasking overhead (context switches and cache misses) and interrupts due to timers and I/O. There are no universally accepted procedures to establish this scenario. In fact, since formulation of an artificial scenario can be quite lengthy (and cause the team to vent considerable emotion), many system integrators now routinely use actual avionics code as a benchmark.

The software execution environment, in the early stages of systems design, can only be based on historical data and the functional breakout described previously. As the software requirements and hardware design begin to mature, this can be refined by simulation and, ultimately, hardware test.

Message scheduling and message latency are two significant I/O parameters. Once again, these can be estimated on the basis of simulations and historical data. The message latency is usually specified in the SRS based on the algorithms used in the software.

The capability to provide accurate estimates for these system parameters is the object of interface analysis. The functional element analysis uses these results to estimate the delays involved in routing information among many nodes within the system. An example follows.

10.2.3 EXAMPLE

The simple communication example shown in Figure 10-2 is the topic of this part. The focus is on the detailed parameterization of the functional

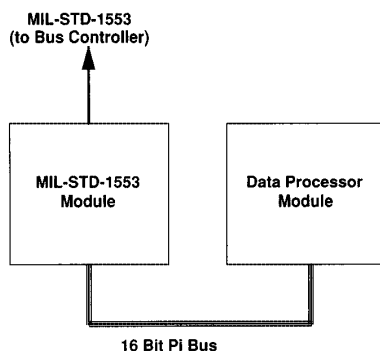


FIGURE 10-2. Simple communication example.

element and interface performance. As such, the example is predicated on a very simple application software program. The impact of performance parameters on system capability, as well as more complex software situations, are topics in the next part.

For this example, two functional elements connect using a parallel Pi-Bus. The first functional element is a data processor module, while the second functional element is a MIL-STD-1553 interface module. This discussion illustrates the impact of the performance and interface analysis on system capability.

Suppose that the software on the data processor module produces a simple navigation output. This might involve numerical integration with respect to time in which the calculations must meet a step size (i.e., Δt) in order to assure the accuracy of the outputs. Therefore, the basis of the SRS requirements is the numerical analysis of the algorithm. More specifically, suppose that the algorithm involves Simpson's Composite rule for numerical integration. Then, the integral with respect to time is given by:⁴ $\int a(t) dt = (\Delta t/2) \cdot [a(t_0) + a(t_f) + 2 \cdot \sum a(t_i)]$, with t_0 and t_f the limits of integration, and $t_i = t_0 + \Delta t \cdot i$. The error in this equation is bounded by $[(t_f - t_0) \cdot a''(\mu)/12] \cdot [\Delta t]^2$, in which $a''(\mu)$ is the maximum value of the second derivative on the interval. In the system, $a(t)$ may be some sort of data input, such as a sensor value which represents acceleration. Therefore, the accuracy needed for the output, velocity in the example, is directly related to the step size and sensor update rate. Since the sensor data ordinarily comes from a special hardware interface, this places a limit on latency for the message containing acceleration. For example, suppose that the desired output velocity accuracy is 1 foot per second and that the error term can be estimated as $a''(\mu) \cdot [\Delta t]^2 = a_{\max} = 9g$. The maximum acceleration of $9g$ ($g = 32.2$ feet per second²) dictates an integration interval of less than 0.0414 seconds. As a result, if this algorithm updates in the system faster than every 40 ms, the output meets the requirements.

Once the requirement is formulated, the system implementor must prove in additional analysis that the hardware and software design can actually perform the update needed in the specified amount of time. For this purpose, a prototype message schedule must be developed. Suppose that the navigation calculation involving the integration requires 5 ms to execute. In the remaining 35 ms, if the integration involves 7 integration points, the acceleration inputs must arrive at least every 5 ms (or faster) at the calculations.

The message latency within the data processor functional element and cluster is the next parameter. To produce this value involves estimation of the software and backplane bus protocol overhead involved with the transfer, specifically the message latency between the arrival of the acceleration input at the I/O module and its receipt by the numerical integration task on the data processor. The I/O module can collect data and transfer it as a block each time a new calculation must be performed, or the transfers can be done in smaller segments. For a Pi-Bus transfer similar to those described in the last chapter, a transfer of the entire block needed for the calculation of the 7 words.

Overhead/message:	18 clocks
Waits/message	14 clocks,
Total data cycles	7 clocks
Waits for data cycles	7 clocks, or

46 clocks at 12.5 MHz, which gives 3.68 μ s. This must be derated by the software overhead, which produces a result in the range of 10 to 15 μ s. A second message contains the velocity output, which the application produces every 40 ms.

In this simple example, The software on the data processor might involve three application tasks: one for I/O, a second for the integration, and an idle task which is a spin loop. The priorities of the tasks are I/O (high), integration (medium), idle (low).

These priority assignments assure that I/O runs whenever new inputs are available and the integration follows such an update. A enumeration variable, SYSTEM_STATE = (I/O, INTEGRATE, IDLE), can identify or trigger changes in system state.

The I/O task starts and then hibernates until the data block arrives, indicated by SYSTEM_STATE = I/O. Upon arrival, the I/O task transfers the data block from I/O memory to application program memory and provides the output message and control information to the interface logic. The I/O task then sets SYSTEM_STATE = INTEGRATE and hibernates again, awaiting the next update, triggered by a timer interrupt.

The integration task triggers when the I/O task sets SYSTEM_STATE = INTEGRATE. If no change occurs, then the integration task hibernates and the system idles. After an update, though, the integration task resumes its calculations to produce an output. Notice that the integration produces the velocity change over a 40 ms interval, but this output is not placed on the I/O channel

until the next 40 ms minor frame. This is typical of sensor input calculation output algorithms which demonstrate a latency of at least two sensor update frames. Upon completion of the integration, the integration task must set `SYSTEM_STATE = IDLE`, then hibernate.

The idle task starts when `SYSTEM_STATE = IDLE`. This task might actually perform module diagnostics or some other system function. As is typical for such applications, they run in the background, which means that these functions only run when nothing else is ready to execute. This is somewhat like scheduling highway repairs during low rate traffic hours, such as at night. The idle task might also monitor the I/O interrupt (or I/O hardware status) and change `SYSTEM_STATE` to I/O when this interrupt occurs.

The basis for the processor throughput analysis for these functions includes memory transfer times for the I/O task, calculation time for the integration task (probably derived from prototype code or historical results), and specified execution times for the diagnostic routines.

The analyst must derate these values for slower memory speed during the I/O activity, as well as two context switches per task activation. For example, for the I/O task, the memory access to the 80 ns I/O memory requires 0.56 μ s for the 7 input words. A similar delay occurs with the outputs of the calculations.

The total execution time of the integration portion of the velocity calculation is:

- 7 data reads at 40 ns (from local memory, not I/O memory), or 0.28 μ s,
- Approximately 7 additional data fetches, or 0.28 μ s, due to cache misses (the I/O data are likely stored in local memory),
- $7 + 5 = 12$ multiplies, 6 adds, one division (see algorithm above) at 40 ns each (one clock cycle per 25 MHz instruction), or 0.76 μ s,
- Two context switches at 100 μ s each.

This provides an estimate of 0.201 ms for the integration. Combined with the previous estimate of 0.003 ms for the I/O task, this means that the idle task runs for over 39 ms out of the 40 ms cycle time, if this were the only calculation.

One additional aspect that an analyst must consider is that a system cannot simultaneously achieve maximum processor and interface throughput. A model which linearly relates these parameters may be an appropriate basis for investigating whether the module can fully meet the performance requirements. The designer must verify that the design can meet reserve requirements throughout a representative portion of this envelope.

The code size estimate is most solid when founded on prototype software for an application which is comparable in complexity to the actual application. On the basis of this estimate, the projected memory reserve emerges.

The numerical integration example is unrealistically simple, but it illustrates some impacts of throughput, scheduling, and latency on reserve requirements. A systems designer would likely not allocate high speed memory and processors to a software application that idles over 99% of the time unless some sort

of high bandwidth asynchronous event must be handled. Also, an aircraft traveling 300 knots represents a 5 nautical mile per minute rate, or 506 feet per second. At a constant speed, this aircraft travels 20 feet in the 40 ms frame time. Typical weapon delivery algorithms assume more accuracy than this, so the system requirements may not correctly relate to operational requirements.

The analysis of the I/O module is similar. In fact, many common module families have identical data processor and base memory configurations. If the modules of Figure 10-2 are of this sort of design, then the same estimates apply as for the data processor module.

In system analysis, end to end latency determines system capability. This means the amount of time delay from the sensor input transmission to the I/O module to the time when the navigation result is received by the destination. Consequently, the analyst must include a second bus side analysis, for the MIL-STD-1553 interface.

MIL-STD-1553, as well as some other bus standards, rigidly defines the maximum permissible response time. For this simple example, the output is a single word message. The word length on MIL-STD-1553 is 20 bits (16 bits data) and the clock rate is 1 MHz. If the cluster shown in Figure 2 operates as a Remote Terminal (RT) and the message is sent to the Bus Controller (BC), the message sequence is:

- BC sends a command word to the RT to initiate the transfer (i.e., BC reads the RT)
- RT response time
- RT responds with a status word and the data word.

The message latency is (a bit time is 1 μ s at 1 MHz):⁵

Control word	20 bit times
Response word	10 bit times
Status word	20 bit times
Data word	20 bit times

It is also customary to add an additional end of message delay equal to one word (20 bit times) for termination of the message. Therefore, in this example of 7 sensor inputs, the bus transmission requires 0.190 ms. For systems with additional messages, the estimate must also include an intermessage gap. The next part provides more detailed examples.

The bus loading is also a common method to parameterize I/O reserves. In this example, since the system schedules the output at 40 ms, this message contributes 0.48% to bus loading.

The software and hardware considered for this example illustrate as simply as possible the estimations needed early in a design for reserves. The next part treats more complex designs at the system level.

10.3 SYSTEM ANALYSIS EXAMPLE

This part explores the impact of node performance and interface analysis on the functional element network, or system. The analysis of storage nodes (i.e. memory modules) is similar to the other module types and is therefore not specifically elaborated. The systems analyst is responsible for early estimates of I/O and processor performance reserves, memory capacity reserves, and bus loadings.

In addition, as part of this analysis, the systems engineer must identify any bottlenecks or single point failures. Other factors, such as data security, may play important roles in the design from the earliest stages.

The key inputs for this analysis and their sources are:

- Functional requirements of the computer hardware (System Specification)
- Functional requirements of the software (SRS)
- Interface requirements of the software (IRS).

The first step in the analysis involves either deriving or obtaining software task schedules and interface message schedules.

From this information, the analyst can derive the output parameters for each functional element and interface, such as message latency, processor throughput needed, and storage space (memory size) needed.

The analyst may derive these parameters on the basis of historical data, analysis of data bus specification timings, examination of compiled code, benchmark results from prototype hardware test, or simulation.

Earlier text provides an informal evaluation of program cost, schedule, and high level technical goals. In this chapter, the analytical techniques of the previous two chapters provide a basis for a more careful proof that the baseline design meets the technical requirements. The exposition of the examples addresses the following specific topics:

- System level message scheduling
- End to end message latency analysis
- System mass storage requirements
- Selection of different modes of operation for system buses.

This discussion addresses some high level performance requirements among various processor types. The analysis includes the system level topics, such as end to end message delays, comments on bottlenecks, and comments on single point failures.

The topic of this chapter is the technology retrofit, an upgrade to an existing aircraft architecture that Part V includes. The previous focus is trade-offs regarding power, weight, volume, schedule, and cost. This subchapter examines the technical performance of the design baseline.

The discussion begins with a description of the system requirements, as well as a summary of the existing system and proposed design. The presentation concludes with a detailed analysis.

10.3.1 SYSTEM REQUIREMENTS

The proposed upgrade involves replacement of the Display Processor (DP) and Mission Computer (MC). There are general computer resource requirements which apply to both, as well as unique requirements for each.

There are five general computer resource issues that the systems designer must face. First, the software must be coded in Ada. In addition, a common software support environment is required, which means that the same Ada compiler and software support tools must be used for both computers. This relates to the second general requirement, which dictates a common data processor for all new designs for the aircraft. The third general issue deals with reserve capacity and growth. Each functional area must provide 50% performance reserves relative to the baseline software needs at the time of delivery for processor throughput, memory size, and I/O throughput. The fourth general requirement is that the software be loadable via the MIL-STD-1553 interface. The fifth general requirement is that the design shall not contain parts which are near or beyond obsolescence.

The primary functional requirement for the display processor is to produce video outputs which drive up to three displays, to support the following subfunctions:

- Aircraft engine and fuel status
- Weapon arming and status
- System diagnostics
- Navigation/waypoints display appropriate to the navigation facilities which the crew selects
- Communications equipment summary report
- Primary flight instruments display in Horizontal Situation Indicator format
- Primary flight instruments display in Vertical Situation Indicator format.

The display processor function must accept inputs from the Mission Computer, calculate the outputs needed for the display which the crew selects, and generate the video signals. In addition, the display processor function must provide a diagnostic operation which monitors the mode and submode of operation of this function.

The MC must provide the following functions:

- Network management, involving message routing and error detection
- System safety checks involving a double check of weapons status, especially the capability to jettison

- Backup modes, in the event of failures to other aircraft computers, which allow degraded modes of operation to include weapons jettison, fuel dump, return to base navigation, primary flight instruments.

The operation of these functions in the existing system is the next topic.

10.3.2 DESCRIPTION OF EXISTING SYSTEM

The existing system is a tactical aircraft with federated avionics. The system interconnect is two MIL-STD-1553 data buses. Two Mission Computers (MC) are Bus Controller (BC) on one MIL-STD-1553 bus and backup bus controller, or Remote Terminal (RT) in normal operation, on the other bus. These computers act as system controllers since all messages are either BC-RT or RT-BC. This means, in effect, that all messages to be sent from one subsystem to another must be sent to a MC, reformatted, and retransmitted to the destination subsystem.

Two Display Processors (DPs) are provided which have RT interfaces on each of the two primary buses. In addition, the DPs utilize RS-170 as a control interface and produce standard raster and stroke color outputs to the Cathode Ray Tubes (CRTs).

Figure 10-3 (Figure 9-29 reproduced here) shows the system block diagram. This architecture is federated, since each avionics function is mapped onto a separate computer.

The system avoids single point failures by duplication of each hardware element. This includes multiple MIL-STD-1553 buses, dual MC, and dual DP. In addition, each computer has software backup modes which permit continued operation if any single computer should fail. On line diagnostics are likely to include loop back tests, in which each computer (MC or DP) sends a diagnostic message to the MC, after which the MC sends an acknowledge.

The mission computers represent a possible system communication bottleneck since each message must transfer to a MC, then cycle to its ultimate RT destination. This issue often becomes important with federated systems whose performance is marginal.

A mature system usually does not meet the 50% reserve capacity requirement. This is ordinarily acceptable since functional requirements are stable for a mature system. However, reserves again become an issue if the addition of new system functions is the motivation for the upgrade. This is especially true if system performance is marginal.

Finally, hardware parts obsolescence may be another motivation for the upgrade. At some point, microcircuits become so old that the original suppliers may stop production, or even go out of business.

10.3.3 DESCRIPTION OF PROPOSED SOLUTION

The proposed solution is the Integrated Mission/Display Computer (IMDC). This approach is similar to numerous retrofit programs which are moving

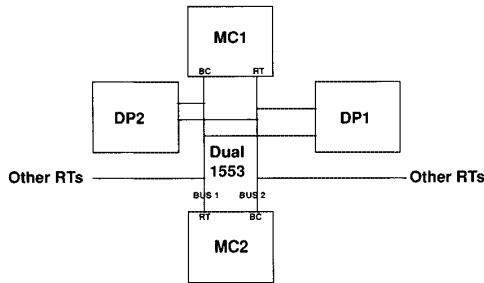


FIGURE 10-3. Tactical aircraft baseline architecture.

slowly to functionally integrated subsystems. Display Processing (DP) and Mission Processing (MP) are the two most common starting points for the functional integration.

The IMDC baseline consists of the following hardware elements:

- Two newly developed data processor modules
- One newly developed special Input/Output (I/O) module
- Two off-the-shelf graphics processor modules
- Two off-the-shelf MIL-STD-1553 modules
- One off-the-shelf power supply module.

Actually, it is possible to combine the graphics and data processing functions rather easily into a single module type. The combined module provides:

- 4 Intel 80860™ processors, programmed in concurrent (multiprogramming) Ada
- 8 MB of memory
- A VME interface.

The I/O module provides the video signal outputs. Since these signals are analog, a detailed analysis is quite lengthy and complex. Therefore, the analysis of the next topic focuses on the digital channels.

MIL-STD-1553 modules provide:

- Two, dual redundant, transformer coupled, MIL-STD-1553B bus interfaces per each module
- A 1 MB buffer memory
- A VME backplane interface.

The power supply is also analog and the subsequent analysis does not include its functional details. However, careful analysis is important because

power supplies historically represent a reliability problem area. On the other hand, a complex engineering analysis of the power system offers little new insight into the design process, which is the topic of this discussion.

The primary timing requirements for the power supply relate to start-up time, sequence time after commanded reset, shut down time after system error. Therefore, the power supply timings for normal system operation are usually not a performance issue, as these timings relate to error recovery or start-up procedures.

The software architecture consists of two Computer Software Configuration Items (CSCIs), one each for the DP and MP functions. This software contains numerous error recovery and backup modes, as with the original. An important difference, though, is the capability to more closely isolate hardware errors and limit the propagation of any erroneous data within the system due to a failure.

The message mix for the navigation display involves 50 data words, which are transferred as 5 messages of 10 words each.

The analysis in the next topic examines the compliance of this design approach with the technical performance requirements,

10.3.4 ANALYSIS OF PROPOSED SOLUTION

The baseline design easily provides sufficient memory and processor throughput capacity. The I/O requirements are less transparent, though. An example of a message latency requirement is an end to end message latency of less than 0.5 seconds from the MP to navigation system and return to display. The value of 0.5 seconds lag time (not video refresh rate) is the visual threshold that may be confusing or misleading to the air crew.

The first requirement relates to the display of navigation information. The navigation subsystem usually provides a simple latitude/longitude output, but the mission computer navigation manager calculates a best guess, based on input blending. These inputs include the inertial navigation unit, and radio navigation inputs such as Global Positioning System. The message latency requirement stems from the need for the air crew to correlate the navigation display with the inertial subsystem output. If this delay is too large, it becomes confusing to the crew whether the navigation manager is functioning properly. In addition, there might be a test mode, which simply passes the inertial solution through the navigation manager to the display, to check the video interface. Once again, excessive delay in the test mode might cause confusion.

The following message scenario describes the communication:

- The navigation manager solicits inputs from the MIL-STD-1553 interface from five other subsystems which contain navigation solutions
- The navigation manager must process these inputs
- The navigation manager sends the solutions to the display processor
- The display processor generates the graphics image and provides this output.

The MIL-STD-1553 inputs are responses to the BC, in this case the navigation manager running within the MP CSCI. Each RT responds when interrogated by the BC. Since the message formats are similar, the BC simply sequentially strobes each RT within its own 50 ms minor frame. Operating in RT-BC mode, if each message contains 10 words, the time required for each message is $40 + 20 \cdot 10 = 240$ clocks per message.¹ Each message is separated by 100 clock cycles of intermessage gap (worst-case), so the total time for the 1 MHz bus is $340 \cdot 5 = 1.7$ ms. In contrast, if RT-RT mode is used to communicate directly (the navigation manager would likely be part of the DP CSCI), then $116 + 20 \cdot 10$ clocks per message would be needed for a total of 2.08 milliseconds for the transfer. Although RT-RT mode requires more bus time, it can sometimes reduce the amount of processing needed and ease system communication bottlenecks.

The navigation manager software within the MP CSCI must perform MIL-STD-1553 I/O with navigation elements, process inputs to produce an output, and provide the output to the DP CSCI using the VME backplane I/O interface.

The MIL-STD-1553 software interface generates considerable overhead due to the need to pack and unpack data words into appropriate binary formats. This overhead consists usually of a procedure call and processing such as shown in Figure 10-4. An example calculation for this overhead is:

- One interrupt service routine time at 15 μ s
- 50 procedure calls (one for each data word) at 15 μ s each, or 650 μ s
- 50 processing times of 0.28 μ s.

This produces a processing time estimate of 779 μ s total. One interrupt service time, not 5, is added because a good systems design would transfer all 50 words in a single block from the MIL-STD-1553 module to the processor module, using multiple MIL-STD-1553 messages.

The DP CSCI resides on the same module as the MP CSCI, so the transfer time is simply a memory copy, or 4 μ s at 25 MHz, two clocks for each of 50 words.

The VME hardware latency occurs during the transfer from the MIL-STD-1553 module to the data processor module. The VME block transfer operation is very similar to the Futurebus+ message passing, which the previous bus side analysis includes. VME allows up to 236 bytes per block transfer, so that each of the five messages of ten 16 bit words (two bytes) can transfer in a single block. In addition, there is one master acknowledge and one slave acknowledge, as well as approximately 150 ns host side set up time. This results in an estimate of $(100 \text{ bytes data} + 2 \text{ bytes acknowledge}) \cdot 40 \text{ ns} + 2 \cdot (150 \text{ ns setups}) = 4.38 \mu\text{s}$. As usual, the bus hardware is fast enough to be essentially zero wait state, but this speed must be derated for software overhead. A factor of 3, is common, which gives about 13 μ s.

```
procedure UNPACK is
  (REAL_OUT : float;           — Output from this procedure
   INT_IN : UNSIGNED;          — Input between +32767 and -32768
   SCALING_FACTOR : float;      — Scales if the output range is not from +32767 to
                                — -32768
   OFFSET : float)             — Offsets if the output is not centered on zero

begin

  REAL_OUT := (float(INT_IN) * SCALING_FACTOR) + OFFSET;

end UNPACK ;
```

=====		
OPERATION	#	TIME (μs)
Type conversion	1	0.12
Floating point multiplication	1	0.04
Floating point addition	1	0.04
Store floating point	1	0.08
Total		0.28

FIGURE 10-4. Sample MIL-STD-1553 unpack routine.

The sum of these factors accounts for an end to end latency of 1,484. However, the host side delays on the MIL-STD-1553 module and within the navigation subsystem must also be included in the estimate. Although processing and memory times are approximately the same as before for these elements, this is not the correct approach since the data processor accesses these resources asynchronously. This means that the navigation sources update their output buffers on 50 ms boundaries, but these minor frames are not synchronized with the data processor minor frames. In addition, the slave software on the MIL-STD-1553 module introduces a similar skew, as does the DP CSCI. In short, at least two minor frame delays must be added to the overall message latency to account for scheduling skew. This accounts for 0.100 seconds, which dwarfs the hardware response times. For some systems, this type of delay is unacceptable and a *system time* can be provided for all modules with excellent resolution.

This example shows that a verification of system requirements must include two distinct types of calculations. The first calculation deals with hardware performance and provides an estimate of how fast the hardware can respond from one end of the system to the other. The second factor is scheduling latency, which accounts for software and message scheduling in the nodes. In general, the software latencies will dominate the overall message latency, but the hardware response time is needed to verify that adequate performance is available. Proper integration, though, usually results in hardware latencies being negligible, but scheduling latencies being important.

REFERENCES

1. **Klein, Mark H. and Ralya, Thomas**, An Analysis of Input/Output Paradigms for Real-Time Systems. *Carnegie-Mellon University/Software Engineering Institute Technical Report CMU/SEI-90-TR-19*, July 1990.
2. **Silberhatz, A. et al.**, *Operating System Concepts*, Third Edition, Addison-Wesley 1991. Chapters 4–6.
3. These are hypothetical numbers, but some actual values can be found in: **Newport, John**, A Common Ada Interface For Message Passing Systems, Naval Air Warfare Center, Indianapolis Technical Report 2280, July 1993.
4. **Burden, Richard L. et al.**, *Numerical Analysis*, Prindle, Weber, Schmidt, 1981, page 149.
5. MIL-STD-1553 Data Device Corporation, Designer's Guide, 1990, pages 1–59.