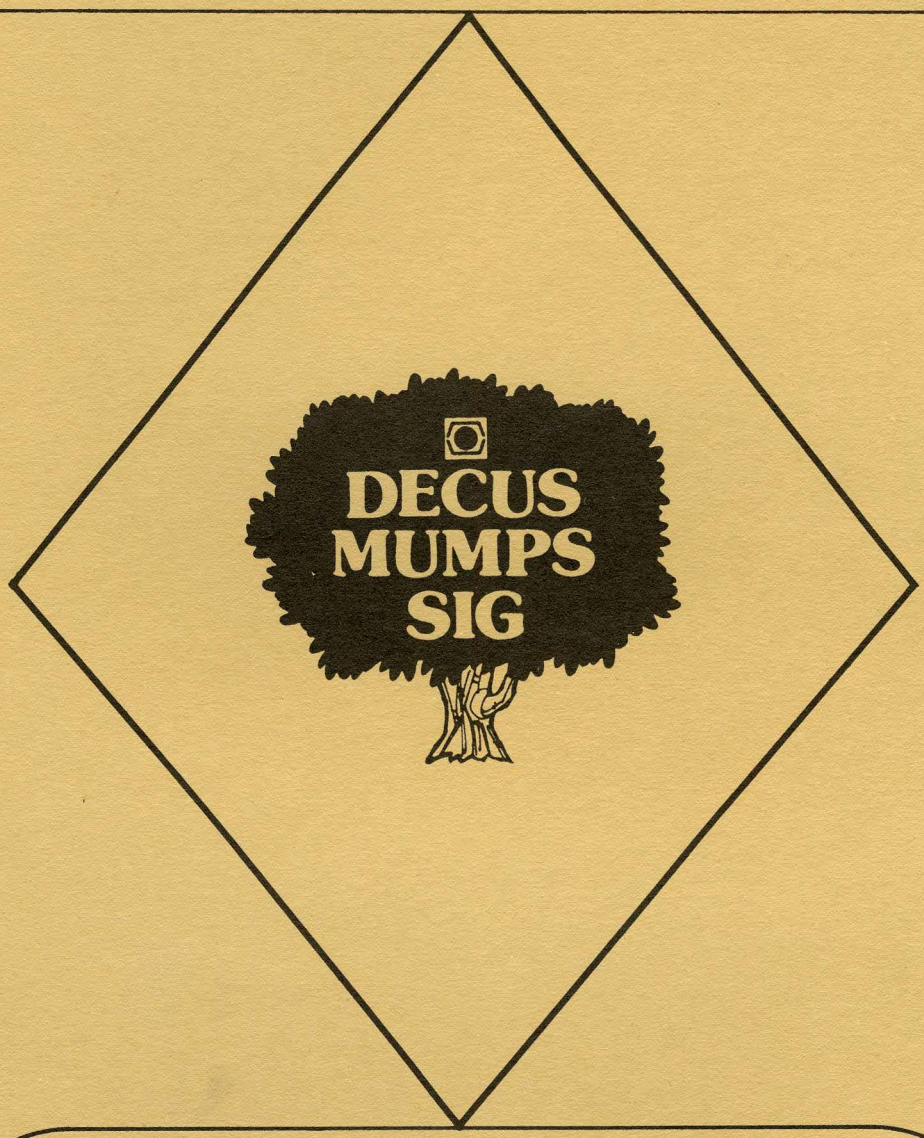
Structured Languages SIG DECUS U.S. CHAPTER

THE HEAP

STRUCTURED LANGUAGES

DECUS



February 1984 Issue

Printed in the U.S.A.

The MUMPS SIG

Did not submit material for this issue

Send your submissions for the next issue to:

Jim Bernard

Data Processing
Kettering Medical Center
3935 Southern Blvd.
Kettering, OH 45429

THE HEAP

From the Editor

John R. Barr, University of Montana, Missoula, MT

This is really the November issue of the Heap which is appearing late due to the lack of submissions and my own personal time. This month's award for letting people know what you are doing has to go to Kathy Hornbach (Lear Siegler) who is our Productivity Tools Coordinator. It looks like Kathy is using her software tools effectivly as she has the time to tell people what she and her people are doing. How about the rest of the people who read this newsletter? Why are you so busy that you can't tell other people what you are doing? There must be more people using software tools and languages that have something to say about them. How about sending me something for the newsletter? I will accept RT-11 floppy disks, UNIX tar tapes (1600bpi only), ANSI standard tapes, DEC20 tapes, or give me a call and I will give you access to one of my systems (300 baud) to type it in. We also have Kermit on the DEC20 to transfer a completed file. Use DSR (Runoff) without any fixed margins. If you have to send me reproducable copy I will paste it in, but I prefer to use DSR as it produces much better copy.

We are planning two pre-symposium seminars for Cincinnati. One by Kathy Hornbach "Implementing a Software Development Environment" and one by myself "Modula-2 Programming and Concurrent Programming Techniques". Let us know if you are interested in other topics for future symposia. I will not know what sessions will be presented in Cincinnati until February. Look for the next issue of the HEAP sometime in April to contain a review of what the Languages and Tools SIG has planned.

A couple of the newer members of the SIG steering committee have submitted an outline of what they will be doing for this newsletter. I will include a complete list of the SIG leaders in the April HEAP.

From the Editor

This issue contains two excellent articles on how Lear Siegler is utilizing productivity tools in the VAX/VMS environment. I am sure you and others can profit from their experiences and may even have something to contribute along these lines. Let me know what you think.....

John R. Barr
Department of Computer Science
University of Montana
Missoula, MT 59812
(406) 243-4807

Modula-2 Interest Area Coordinator

I am Jack Davis, the LTSIG Modula-2 Interest Area Coordinator. I am attempting to compile a resource list for Modula-2: sources of information about the language, sources of compilers that generate code for DEC machines (from PC's to 10/20's!), or run on DEC machines generating code for other processors, and also information about who is working with Modula-2 to enhance existing implementations or produce new ones. I would also like to be informed of useful tools and other programs that have been implemented via Modula-2. Last but certainly not least, I would like to hear from individuals who have encountered and/or fixed bugs in Modula-2 implementations.

The information I collect will appear in a later edition of the HEAP.

For those with questions or information to contribute, my telephone number is (615) 690-3160 (eastern time), and my address is

Jack R. Davis
NAP Consumer Electronics
9041 Executive Park Drive
Suite 612
Knoxville, Tennessee 37923

UNIX Coordinator

As the new UNIX coordinator of the Operating Systems interest area of the LTSIG steering committee, I would like to introduce myself. My name is Rod Creason and I work as a Systems Programmer and Project Leader for Compiler Design and Development at Digital Information Systems Corporation (DISC). My address is:

Rod Creason, Jr.
Digital Information Systems Corporation
3336 Bradshaw Road 340
Sacramento, California 95827
(916) 363-7385

UNIX Coordinator

My function as UNIX coordinator will be to track and Languages and Tools associated with UNIX, test and document any new UNIX contributions, act as liaison to the UNISIG, and work closely with the Operating Systems Coordinator of the LTSIG. Please get in touch with me if you have any thoughts which might concern my area.

RT-11 Operating System Coordinator

I would like to introduce myself to the Languages and Tools members as the new RT-11 operating system coordinator. My name is Michele Wong. I am the manager of Software Development at Digital Information Systems Coorporation (DISC). At DISC I am responsible for overseeing the development of the DBL language. Most of our work is systems-level work on RT-11, TSX-PLUS, RSTS, RSX-11M, and VAX/VMS.

AS RT-11 coordinator for the LTSIG, I will be responsible for communicating information relating to languages and tools under the RT-11 operating system between the Languages and Tools and RT-11 SIGs. I will also be providing support to RT-11 users in the languages and tools areas. I can be reached at the following address:

Michele Wong
DISC
3336 Bradshaw Road, Suite 340
Sacramento, California 95827
(916) 363-7385

I am looking forward to working with the members of the LTSIG.

Toolside Chat

This is the first of a quarterly column, answering questions on software tools and methodologies. It is written by Kathy Hornbach, from Lear Siegler in Grand Rapids, MI. If you have a question on software tools or methodologies you would like answered, send it to:

John Barr
LTSIG Newsletter Editor
Computer Science Dept.
University of Montana
Missoula, MT 59812

This month's question:

How can I find out what software packages are available for DEC machines - both from DEC and from 3rd party vendors?

Answer:

A good place to start is with some literature provided by DEC and DECUS - they have several books available that contain a host of software packages. The books available include:

- Software Referral Catalog, 10th edition; from the Engineering Systems Group.

  It contains over 600 entries, under the headings of chemical, civil, earth resource, electronic, mechanical, optical, power systems and structural engineering; also CAD, CAM, engineering libraries, general engineering tools, and management and administration. Packages listed run on a variety of DEC machines.

  To receive future editions of this catalog, write to:

  Digital Equipment Corporation
  Software Referral Catalog Attn: SRC Manager
  Engineering Systems Group MR03-1/E8
  2 Iron Way
  Marlboro, MA 01752

- PDP-11 Software Source Book, 1st edition;

  Contains 980 pages of software that runs under the various PDP-11 operating systems, in all areas - from business to engineering to languagues.

  To obtain a copy of this catalog, contact your local Digital Sales office, or write to:

Attn: G. Deforge
Printing and Circulation Services
Digital Equipment Corporation
444 Whitney Street
Northboro, MA 01532

and ask for PDP-11 Software Source Book, Order No. ED-24762-20.

- o UNIX* Software Guidebook, First Edition

  Contains 180 pages of software packages that run under the UNIX operating systems. Both applications and systems software packages are listed. To obtain a copy, contact your local Digital Sales office, or write to:

  Attn: G. Deforge
  Printing and Circulation Services
  Digital Equipment Corporation
  444 Whitney Street
  Northboro, MA 01532

  and ask for UNIX Software Guidebook, Order No. EJ-25541-20

- o Graphics Referral Catalog, third edition. From the Engineering Systems Group

  Contains 75 pages of information about graphics hardware and software that is available for a variety of DEC machines. To receive future editions of the Graphics Referral Catalog, write to:

  Digital Equipment Corp.
  Graphics Referral Catalog Attn: GRC Manager
  Engineering Systems Group MR03-1/E8
  2 Iron Way
  Marlboro, MA 01752

- o U.S. Chapter DECUS Program Library Software Abstracts,

  Over 150 pages of abstracts of user-donated sofware that is available for order from the DECUS library. You must be a DECUS member to use this library (you also must be a DECUS member to subscribe to this newsletter, so you should not have any problems). To get a copy of this catalog, write to:

  DECUS Publications
  MR02-1/C11

*UNIX is a trademark of Bell Labs

One Iron Way
Marlboro, MA 01752

All of the above are invaluable resources when someone stops by and asks if there is any software available to ".....". They are frequently revised to incorporate new packages. Later articles in this column will talk about non-DEC sources of tools catalogs.

USING CMS FOR A VERY LARGE SOFTWARE PROJECT

Robert Gable
Lear Siegler, Instrument Div.
4141 Eastern Ave. S.E. MS 121
Grand Rapids, Michigan 49508

"When the going gets tough, the tough turn to code management."
-- Raoul Duke

INTRODUCTION

We have spent the last 6 months setting up and using a source code control system for a large, real-time avionics software project. This software is being developed and initially checked out on a VAX-11/782 running VMS and a TI 990 minicomputer running DX10. The following will attempt to share some of the experiences we have had in using DEC's Code Management System (CMS) on the VAX for our source code management system.

PROJECT BACKGROUND

Lear Siegler is currently developing the hardware and software for a state of the art flight management computer system (FMCS) be used in a large commercial aircraft. The system can fly a plane on airline routes from origin to destination automatically and in the most fuel efficient manner. The system consists primarily of three computer processors; one for navigation, one for performance, and one for the I/O. We are using CMS to manage the operational flight program which consists of over 300,000 lines of source code for the three processors.

The people concerned with the source code on this project can be divided into three groups:

- development

  These are the people who write the design documents, write, debug and module test the code, and who integrate all of the modules after they have been coded until they are ready to undergo system test.

- verification

  These are the people who approve or disapprove module tests from designers, test the code by individual function and do hardware/software integration tests.

- support people

  This is everybody else.

The delivery schedule to the customer has been tight, meaning that development, integration and verification of the software is all being done concurrently. Development engineers, integration engineers, and verification engineers all need access to different versions of the FMCS source code. CMS provides the necessary common environment for storing and retrieving that source code.

Development is being done in three "packages" or releases where each package has more functionality than the previous package. Because of the tight schedule, more than one package is being developed at a time. The later package uses some of the previous package's code untouched, modifies some of the previous package's code to add functionality, and also adds new modules of its own. This means that while a module is still being developed for the earlier package, sometimes it must at the same time be modified and used for development of the later package. Eventually, different versions of modules between packages must be resolved. More about this concurrent development later.

To attempt to control this type of source code development using the more informal ways of the past would have been impossible. Therefore, we have chosen CMS to be the mechanism to manage our source code in a controlled, yet flexible manner. As an added benefit, we can generate data using CMS that attempts to quantify the progress of the software.

USAGE OF OUR CMS LIBRARY (SO FAR)

Most of the following information is derived from the CMS history file.

- ~2200 source files for a total of ~60000 blocks

- ~50 people use the library

  This includes development engineers, verification engineers, clerks, summer students, configuration management and software quality assurance personnel and even project managers.

- over 200,000 library transactions in 6 months

- o typical weekday usage

  The library gets heavy interactive use during the day and large batch FETCHes and INSERTs during the evening. Users were given plenty of encouragement to delay their big batches until after peak hours and they have cooperated.

  - ~1500 transactions a day (the peak day so far had 3000)
  - 50-200 RESERVEs or REPLACEs a day

- o the most frequent commands in order are:

  FETCH, INSERT, RESERVE, REPLACE, CREATE, REMOVE, UNRESERVE

- o we backup the library twice a day (6AM, 6PM)

- o 20% of the users have 80% of the library usage

HOW THE LIBRARY IS SET UP

- o all modules are in only one library

  This was necessary because some modules are used in more than one processor.

- o source modules consist of FORTRAN code, assembly language code and common blocks that are lexically included at compile time

- o common blocks are treated as multi-file elements

  Our common blocks are generated automatically by a Datatrieve procedure that works automatically from our "design dictionary". Both assembly and FORTRAN versions of every common block are generated. We store them in CMS as a single element. Everytime you FETCH, RESERVE or REPLACE a common block from the library, CMS automatically either gives you a copy of both versions or stores both versions back into the library.

- we have the history attribute enabled

  Unless you specify otherwise, the CMS replacement history is automatically appended to the end of source modules copied from the library.

- a source module is uniquely identified by the its CMS element name and generation number

  Earlier in the project, we had another method of identifying modules but element name and generation number turned out to be the superior way of identifying modules for module test status lists, software configuration management paper documents etc.

PARTIAL LIST OF CODE MANAGEMENT TASKS

The following are the standard ways in which project engineers interface with the library.

- development engineers - adding a new source module to the library

  Each development engineer is responsible for putting new modules into the library using a procedure developed to standardize and automate this task. The module must compile cleanly but need not have undergone module level test to be entered in the library.

- development engineers - fetching a read-only module

  This is simply an engineer needing a copy of a module with no intention of modifying it.

- development engineers - problem fix

  This involves RESERVEing a module, revising and debugging it, REPLACEing it back into the library and eventually submitting a completed module test report. The remarks field will contain the reason for the change and usually the problem report number that this change fixes. Some developers FETCH a module, modify it and when debug is complete, do a RESERVE, delete the module just copied from the library, and REPLACE with the module they had modified after the FETCH. The developer must take responsibility for any problems that might crop up by circumventing the "correct" procedure.

We slowly realized that it is better to get the source changes into CMS as soon as possible rather than strictly enforcing the "rules" and have engineers refuse to use CMS until absolutely necessary. One of the most difficult aspects of introducing CMS on this project was deciding on what the guidelines are for when a module can be changed in CMS and when it cannot.

- integration engineers - fetching source to build a test executable for integration work in the lab

  This is where we gain the greatest benefit from using CMS. Integration engineers are required to fetch all of their source used for a build from the library while at the same time recording the exact element and generation number in a CMS class. This is to insure that at any time, what is being run in the lab has been clearly identified in CMS and is reproducible at any time in the future. The engineers quickly realized that it was to their advantage to do this diligently. Eventually, these classes are used to determine what to deliver to the customer.

- verification engineers - fetching source for functional or hardware/software integration test

  After a few iterations, we developed a method whereby one person performs all of the source fetches for verification. The verification engineer informs this person of what he or she desires and the "test fetcher" takes care of all the rest. The engineer provides a list of modules with exact generation number. This list is then used to fetch the modules from the library at an appropriate time and the list is stored elsewhere. At any time until the end of the project, what source was tested will be identified exactly by this list.

- verification engineers - maintenance of module test information

  Verification maintains a progress data base which contains the test status of every module. Among other things, the generation of every module that has passed module level test is maintained in a data base and in CMS classes.

- configuration management, software quality assurance personnel -

Both maintain classes which satisfy their requirements for properly identifying the source that has been tested and released. On previous projects, an extremely cumbersome and error prone manual, paper based system was used. CMS automation has saved many, many hours in this area as well as dramatically improving quality.

- support engineers - overall maintenance of library

  This involves correcting mistakes made by engineers (typos during new module creation being the most common), trouble shooting problems, renaming modules and investigation of better methods of utilizing CMS.

- support engineers - build tools or educate users for more productive use of CMS

  CMS is relatively easy to use so education has been minimal. We wrote an FMCS CMS user's manual to accompany the DEC/CMS manuals. The manual describes how to use the FMCS CMS library, what is the significance of all the classes, what are the naming conventions, what tools are available and how to use them etc. We are on the constant prowl for ideas for tools that will exploit the wealth of information gained by using CMS.

- facilities personnel - insure proper BACKUPs of library; installation and control of CMS

  These are the normal system manager tasks. It pays to be on the good side of these people since CMS does use up resources and the library is a valuable item which needs to be preserved.

CMS CLASSES WE MAINTAIN

- function classes

  The FMCS source code is partitioned into functions. A class which contains all modules constituting a function is created for each function. For example, FPN is the class which contains all modules for the "flight plan management" function in the navigation processor

- integration build classes

  The development engineers responsible for debug integration of the software (i.e. taking software into the lab and doing their magic), create and maintain a class for each build they do, by processor and package. Before they can test a module in the lab, they insert the module into the appropriate class and then fetch it. These are the most important classes since they are the definitive record of what software is being tested. Monitoring the activity of these classes gives a good indication of the amount of development work taking place.

  For example, NAVPK2OCT25 is the class containing all of the modules (with proper generation number) for the navigation processor build for package 2 release on October 25th.

- in-module-test and passed-module-test classes

  These classes are maintained by verification. Every module must have a corresponding module test packet submitted for it by the developer. When verification receives the test packet, the module is placed in the in-module-test class. When verification approves the test packet, the module is placed in the passed-module-test class. By comparing these classes with the preceding integration classes, we can determine which modules being integrated still require an initial unit test or retest.

- software quality assurance passed module class

  This class contains all modules which software quality assurance has found satisfactory.

- software configuration management release classes

  This class contains all modules which have been released and sent out to the customer in a particular release.

MODULE STATUS

By simply doing a CMS SHOW ANCESTOR /CLASS, we are able to determine the following things about an FMCS module. On previous, manual-based projects, most of the information was difficult, if not impossible, to obtain.

- how many generations exist
- which version of a module is the "latest & greatest"
- for each generation:
  - who made the change
  - when they did it
  - what problem reports this change fixes
  - reason for the change
- what processors use the module
- the module title
- the module part number identifier
- what generations have been released to the customer
- what generations are currently in integration
- what generations have been module tested

ADVANTAGES WE GAINED BY USING CMS

- advantages for development
  - accurate access to "new & improved" code for debug work
  - knowledge of what code is currently in integration, what code has been module tested, function tested and system tested and what code has been released
  - source code under visible control

- reservation mechanism to avoid concurrent modification by more than one person (which can be overridden if necessary)

o advantages for verification

- clearly identified record of tested source
- knowledge of debug changes
- source code under visible control
- ability to monitor current status of various packages
- latest version plus all previous versions always accessible

o advantages for project managers

- little time lost due to test of wrong/unknown versions of software
- accurate information on project status and module status

  Most important, this status is available in a timely manner.

- history of source code, available for analysis of metrics, bug origins etc. for later projects

o advantages for the persons responsible for controlling the source code

- CMS is sufficiently flexible to allow evolution in the way we choose to use it
- no human intervention is necessary between the development engineer and the source code library (with a few exceptions)
- data integrity

  DEC has done an outstanding job of insuring that no data gets lost. We have had disk crashes, system crashes, a power glitch, hostile and incompetent users; none of which has caused any source code to be lost.

PROBLEMS WE HAVE HAD

- no "clean" way to turn off access to the library

- no explicit provisions made for allowing BACKUP to run while engineers are accessing the library

  We excluded the library from the normal system backup and do our own backup twice a day. We require that everyone stop using the library at this point.

  The only time we had to restore the library from backup was when the disk went bad 20 minutes after preventive maintenance had been completed.

- response time gets very slow when several users are trying to access the library at the same time

- when an element belongs to many classes, it is very painful to have to rename the element

  It must be deleted from all old classes under the old name and added to those same classes under the new name.

THE PEOPLE

The following are some thoughts about the social and psychological implications about using CMS on a large project.

- Nobody ever had any problem learning how to use the basic commands.

- The development engineer who spends his time coding and testing his own modules is usually the one who objects to using CMS.

  Some of this is probably frustration at having yet another new tool/methodology to have to master.

  When using CMS interactively on a busy day, time can be wasted waiting on CMS commands. yet, few managers viewed the time wasted as significant.

A few people have the attitude that "real" programmers don't need code management systems.

- 95% of all complaints center around response time on a loaded system when several others are using CMS also.

  Getting several "Your library is in use" messages at your terminal can be frustrating. As CMS response time improves, complaints should drop dramatically. Many engineers have become adept at SPAWNing CMS commands and setting up batch jobs rather than doing their CMS work interactively.

- I prefer to see each development engineer accessing the library him or herself rather than delegating that activity to support people.

  The goal is to get any new or changed source in CMS as soon as possible. Otherwise, very quickly you lose all hope of tracking that module. For example, in one case an engineer did not have time to put his modules into CMS himself. Two other people tried to put the same modules into the library (different copies of course). Fortunately, CMS leaves a record of all important transactions.

  As the system evolved, it became obvious that this did not hold true for verification engineers and that it was more efficient to have one person to do the big fetches of source for test

- Both project management and line management at LSI quickly recognized the benefits of CMS and were supportive when problems arose.

  Both they and engineers get the credit for the successful use of CMS. Plus, everybody's complaints and grumbles tended to be humorous in retrospect.

SURPRISES

- Development engineers are reasonably precise about what they put for the remark when doing a REPLACE.

- Development engineers are nearly unanimous in complaining about the response time for CMS commands when several people are accessing the library.

- Multiple batch jobs running CMS concurrently tended to shut out people running CMS interactively.

- Using classes is a boon in tracking development's progress.

- Considering that we have run CMS hundreds of thousands of times, we found only 3 minor bugs that need to be SPRed.

- Variants of the same module for different packages allow concurrent development of the same module, but signal everybody to pay particular attention to these modules to insure proper coordination.

  Allowing variants to evolve from a module is a far cry from 2 or 3 people copying the same module to their own personal directory with nobody knowing that somebody else is working on that module. We originally viewed variants as being too risky and complicated to use but gradually came to accept their use in legitimate cases.

- Despite threats of bodily harm, people continue to delete CMS batch jobs from the batch queue while they are running.

  This leaves the library in an inconsistent (though easily recoverable) state.

- When the library is in an inconsistent state, anyone using CMS gets a message saying "Please use CMS VERIFY/RECOVER."

  Even though we tell people not to do it, they, without fail, never disobey the computer. One time, a project manager and a part time summer student both tried to run the RECOVER at the same time. The student probably did not know better. The manager knew better but did it anyway. Recovery on a library of this size can take 2 hours of exclusive access to the library and is sometimes not necessary. Project tends not to appreciate this down time. Rumor has it that the error message will be changed in the next release.

TOOLS DEVELOPED TO AID IN USING CMS

- tool to put new code into the library

  This tool does a CREATE ELEMENT with standardized remarks, inserts the element into a function class and records some other data

- tool to fetch code from the library for verification testing and record exactly what was fetched

- tool to find compare two classes to find elements with the same generation number that are in both classes

  For example, often we want to know which elements currently in integration have been module level tested.

- tool to record gripes about CMS

- tool to put the latest version of the common blocks into the library

  The tool does RESERVE/REPLACEs only for those common blocks that have changed in the design dictionary since the last time the tool has been run

- tool to show which modules for a package need retest or have passed module test or are currently in module test or have an inconsistent status that needs to be resolved

- the next obvious tool to build is a dependency file generator to allow MMS to rebuild the system from the library

SUMMARY

CMS has proven to be a necessary component of our software development methodology. It provides not only a mechanism for capturing source code changes, but used properly, it allows the monitoring of development and verification progress during an ongoing project. The only technical drawback to using CMS is performance. It has improved with version 1.1 and should improve greatly for version 2.0. Nevertheless, we feel strongly that CMS was the right choice for our source code management system.

If you have any questions about using CMS, I would be glad to answer questions through the newsletter.

A STRUCTURED ANALYSIS METHOD FOR LARGE, REAL-TIME SYSTEMS

Derek J. Hatley
Lear Siegler Inc., Instrument Division,
Grand Rapids, Michigan.

22 Nov 83

1.0 Introduction

In the spring of 1982 we, at the Instrument Division of Lear Siegler Inc. (LSI/ID), were faced with a dilemma. We develop and manufacture real-time avionics systems and, as those systems became more software intensive, larger, and more complex, we had been experiencing all the classical problems: schedules were unacceptable, costs were high, and it was increasingly difficult to get the systems to work as intended. Against this background we were about to embark on another large, complex, and critical project: the Flight Management Computer System (FMCS) for the new Boeing 737-300.

It was clear that, to be more cost-effective, we needed to make significant improvements in our management of requirements, design, and testing, each of which had been handled somewhat open-loop until that time, and we set out on an evaluation of structured methods, hoping to find some which would be suitable for our type of system and which would provide the improvements we were looking for. Our conclusion was that the Yourdon methods had the most potential, but that there was nothing available which would completely meet our needs. Consequently, a decision was made to go ahead and develop our own methods, based on those of Yourdon, but modified and/or extended as needed.

A major part of that decision was the formation of a "Methods Team", with representatives from Engineering Management, Systems Engineering, Software Design, Software Testing, and Support Software. The Methods Team was tasked with:

- developing structured methods suitable for LSI/ID applications
- teaching the methods in-house to all personnel who would need to use them
- investigating the availability of automated aids to support the methods, and develop aids in-house if necessary

- supporting the initial implementation of the methods on the FMCS project.

The results to date of the Methods Team's work are:

- a real-time Structured Analysis method, based on Yourdon's but with a completely new extension to handle the control requirements

- a Structured Design method, also based on Yourdon's, but with some modifications to meet the needs of large real-time systems

- a cross-reference data-base, linking all the primitive statements in the requirements, design, and test documents

- some basic software aids to support these tools

- in-house classes on the methods, with our own texts, and over 200 people now trained in their use

- most importantly, an FMCS which is now well into system test with significantly fewer problems and better quality than any of our comparable previous systems at this stage of development.

This article gives an overview of FMCS and describes the SA method developed for it.

2.0 Overview of the 737-300 Flight Management Computer System

As the name implies, the FMCS manages all aspects of the flight of the aircraft. It interfaces with all other major systems on board and with the pilot and first officer through dual control/display units (CDUs). There are 13 other systems with which it currently interfaces, and provision for three more in the future. These systems include the auto-pilot and auto-throttle, with which FMCS forms part of the outer control loop, so response times are critically important for stability reasons. Typical response time requirements are 50, 100, or 200 msec. Major functions which the system is required to perform are: navigation, aircraft performance calculations, management of the CDUs, management of the flight plan, flight profile prediction, vertical guidance and steering, lateral guidance and steering, and built-in-test. It contains two large data bases: the Navigation Data Base, which contains all the information on waypoints, navaids, airports, airways, standard routes, altitude restrictions and other data for the area covered by the user airline; and the Performance Data Base, which contains numerical models of the aerodynamics of the aircraft and of the engine thrust characteristics. The size of the Navigation Data Base ranges from 96k to 192k words and that of the Performance Data Base from 4k to 12k words (actual sizes

are determined by customer option and aircraft configuration). The size of the executable code is about 200k words (all words are 16 bit).

The response of the system to a given stimulus is highly dependent on past events, current conditions, and predicted future events. Such considerations as the recent history of navigation signals, the current flight phase, and whether the aircraft is approaching an altitude or speed restriction, all have an impact on various aspects of system response.

The CDU has a 14 line by 24 character CRT display, a full alpha-numeric keyboard, a variety of special function keys, and several annunciators for alerting the crew to unusual conditions. Many different displays are available to the crew, and through the CDU they are able to monitor such things as: the progress of the aircraft throughout the flight; predictions of time, distance, and fuel reserves to the destination and alternate destinations; and the status of FMCS and other systems on board. They are also able to enter many kinds of data, including changes to the current flight plan and construction of completely independent flight plans for future use. Figure 1 illustrates the layout of the CDU.

Systems of this type can provide capabilities which had not previously been possible. One of these is to guide the aircraft along a "great circle" path (the shortest distance between two points on the surface of the earth), which is characterized by a continuously changing heading and bearing. Conventional guidance methods use a fixed heading or bearing for each flight leg.

A prime consideration in the development of commercial avionics systems is the fact that they must be subjected to the very exacting requirements of FAA certification. These requirements include, amongst many other things, demonstration of the fact that the system will not fail in ways which will impair the continued safe flight of the aircraft, and that it will not present false or misleading information to the crew. The need to demonstrate these characteristics in systems as complex as FMCS makes it almost mandatory to present all the requirements and design data to the FAA in an orderly and structured manner.

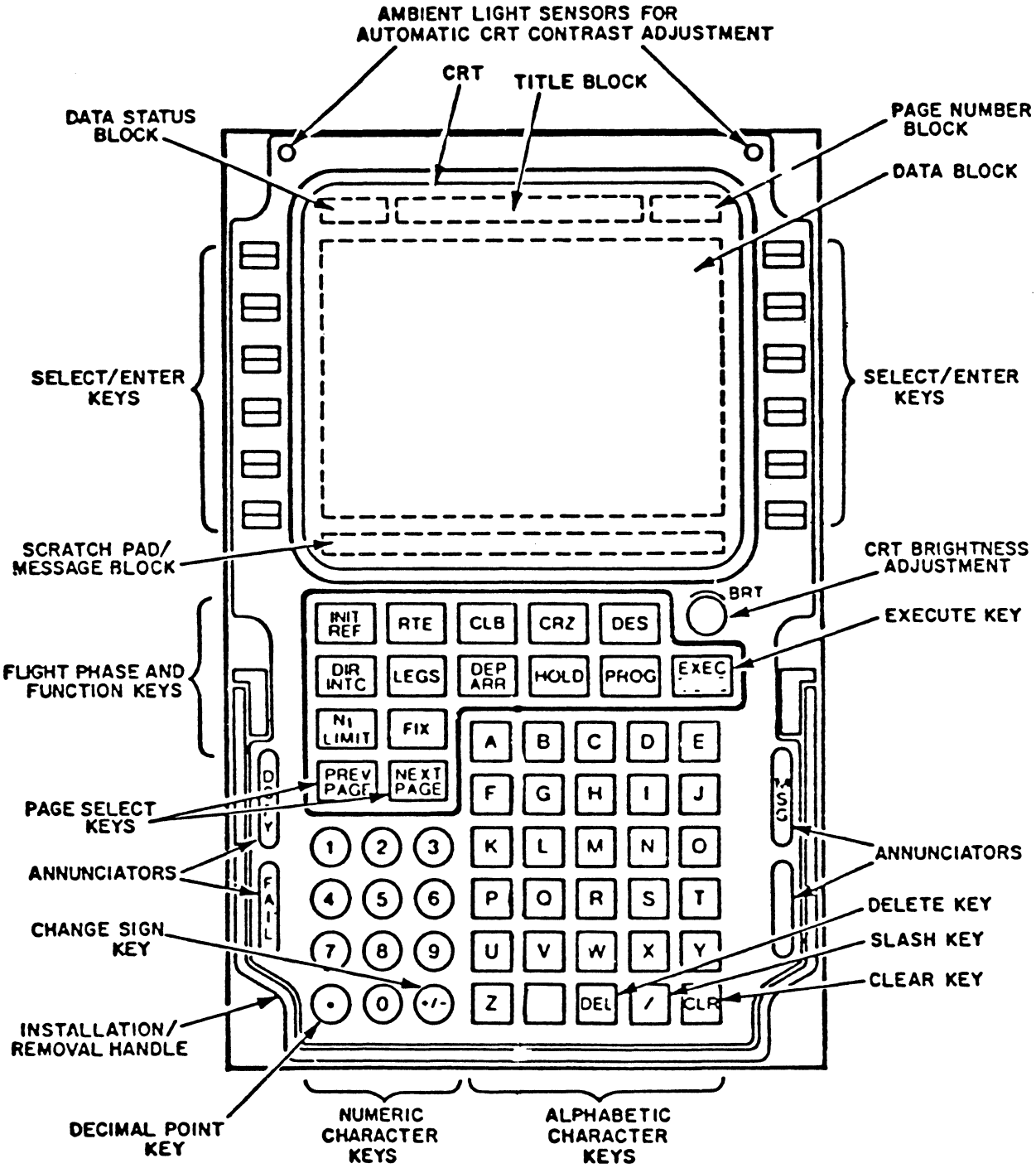


FIGURE 1 - Illustration of FMCS Control/Display Unit

3.0 Goals of the Real-Time SA Method

Recognizing the effectiveness of the basic Yourdon SA method within its own scope, we adopted the goals of that method for the extended method, namely:

- Rigor
- Completeness
- Understandability
- Changeability and Maintainability

In addition, we wanted to minimize the changes to this basic method, and to adopt as many of its features into the extended method as possible. In this way we would take best advantage of all the experience invested in the basic method and make the transition from one method to the other as easy as possible.

Basic SA is an elegant concept, but its strengths lie also in its practicality which arises from features such as leveling, balancing, its numbering system, and the diagrammatic representation. We wanted to adopt these features as much as possible in the extended method, and we also recognized the large body of knowledge available in finite state machine theory and wanted to take advantage of that too.

4.0 Description of the Method

4.1 Overview

There are a number of characteristics which distinguish real-time (RT) from non-real-time systems, two of which are particularly important to this method. First, RT systems contain two distinct types of signals - as well as the familiar data signals (data flows) which are used within data processes, there are other signals, both external and internal, whose primary purpose is to modify the response of the system to incoming data rather than to be processed by it. Second, RT systems are required to recognize past events, current status, and expected future events and, again, to modify system response accordingly.

These characteristics give rise, directly, to the two principle new features of the real-time SA method. First, signals are divided into two types - data signals and control signals - with flow diagrams similarly divided into data flow diagrams (DFDs) and control flow diagrams (CFDs). (Note that the latter are <u>not</u> state transition diagrams). Second, a new type of spec is introduced - the control spec - which represents the finite state (FS) machine characteristics of the system (and which may contain state transition diagrams). To distinguish them from control specs, mini-specs are renamed "process

specs". In fact, several minor changes in terminology have been adopted and are listed below:

Real-Time Method	Basic Method
Data context diagram	Context diagram
Control context diagram	None
Data flow diagram	Data flow diagram
Control flow diagram	None
Process spec	Mini-spec
Control spec	None
Timing spec	None
Requirements dictionary	Data dictionary

Having decided to separate signals into two types it becomes necessary to define how to make that separation in practice. As usual, there are no absolute rules, but some guidelines were established. Any signal representing a continuous physical quantity must be categorized as data. Discrete-valued signals are not always so easily dealt with. The best approach is to refer back to the original principle: if a signal is used within a process as part of a calculation, categorize it as data, if it is used to modify the response of the system to other signals, categorize it as control. It sometimes happens that a signal is used for both purposes, in which case it is categorized as both, and appears both in the DFDs and CFDs.

The primary purpose of the FS machine attributes of the system is to modify the response of the system according to past, current, and expected future conditions. It does this by controlling processes (that is, activating and de-activating them) and can conveniently be thought of in the same way as a feedback control loop in control system theory. Figure 2 illustrates this concept. A second purpose of the FS machine is to signal the status of the system to other systems, and this is done through the control outputs shown in the figure.

Two additional terms are introduced in figure 2: "process controls" and "data conditions". Process controls are the signals which activate and de-activate processes in the data processor, and data conditions are control signals derived through tests on data; for example:

```
If ALTITUDE > 18000ft.
        set HIALT = TRUE
```

in which HIALT is a data condition.

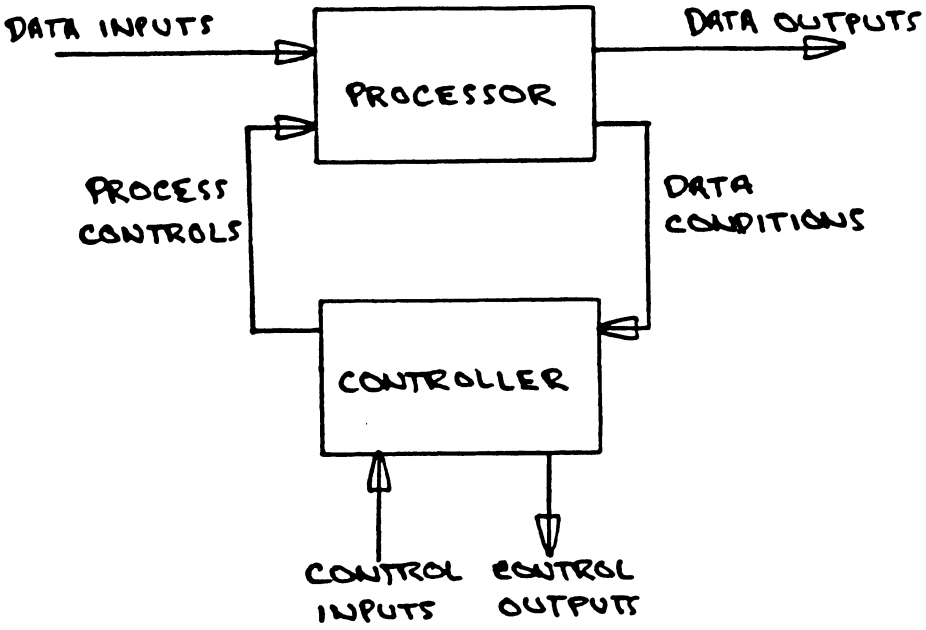


FIGURE 2 - Feedback Control Representation of Method

4.2 Integration of the Control Method into the Basic SA Format

It is characteristic of control system design that the entity to be controlled is defined first, since only then can the controlling mechanism be defined. For example, in designing a feedback power amplifier, the output stage must first be designed to drive the required load, then the number of stages and the loop transfer function can be calculated to suit the requirements of the output stage. This principle was used in structuring the real-time SA method. Since the main purpose of the FS machine is to control the data processor, its structure is slaved to that of the data flow structure. Specifically, control signals are constrained to flow only along the same routes as data signals, and each control spec is associated with one and only one DFD - the one whose processes it controls. This means that each CFD must correspond with a particular DFD and must have the same name and number as that DFD, and that each process on that CFD must have the same name and number as a process on the corresponding DFD. It also means that a control spec must have the same name and number as its corresponding DFD. This gives rise to very tightly coupled groups of diagrams: a DFD, a CFD, and a control spec, all with the same name and number. All the inputs to the control spec come from the corresponding CFD and the two must balance. All the outputs from the control spec are either activators of processes on the corresponding DFD, or new control signals which go directly to the corresponding CFD and must balance with it.

This structure has the very desirable effect of concentrating control requirements close to where they are used, yet there is no loss of flexibility, as the control signals from which the control functions are derived may flow within the structure in just the same

way data signals do. The control requirements simply get partitioned in the same way as the processing requirements are partitioned.

Figure 2 may be thought of as being repeated at each level in the structure and the "controller" block divided into CFDs and control specs. Figure 3 illustrates one level of this configuration.

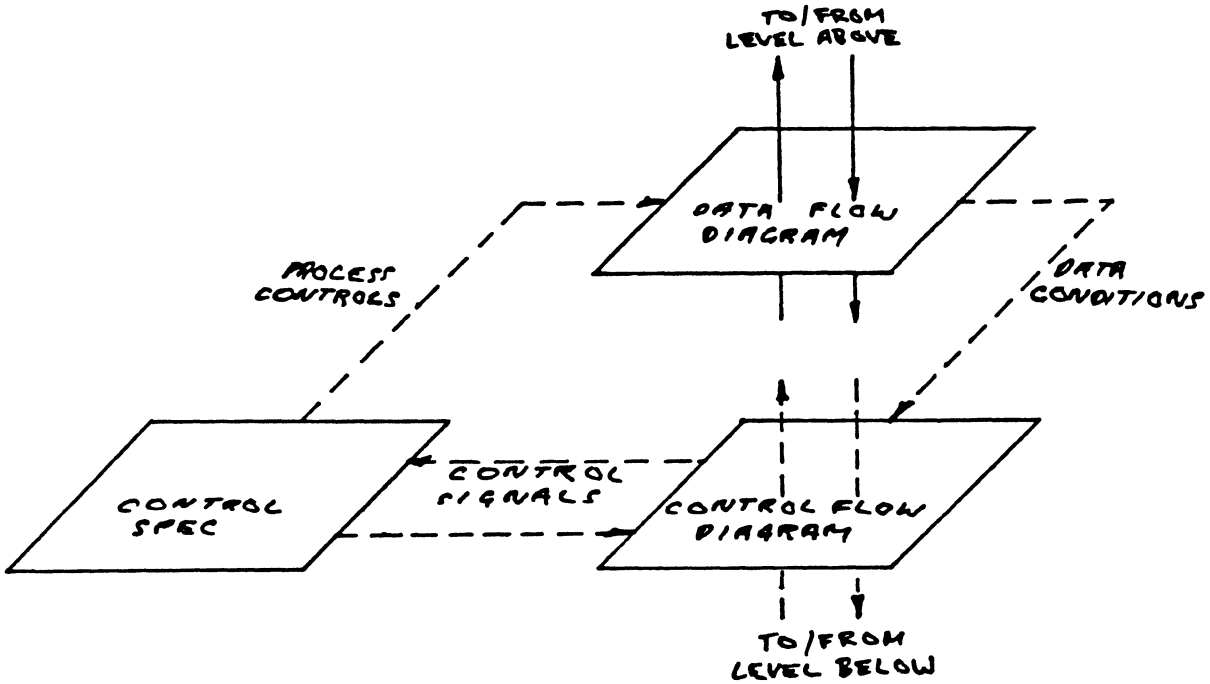
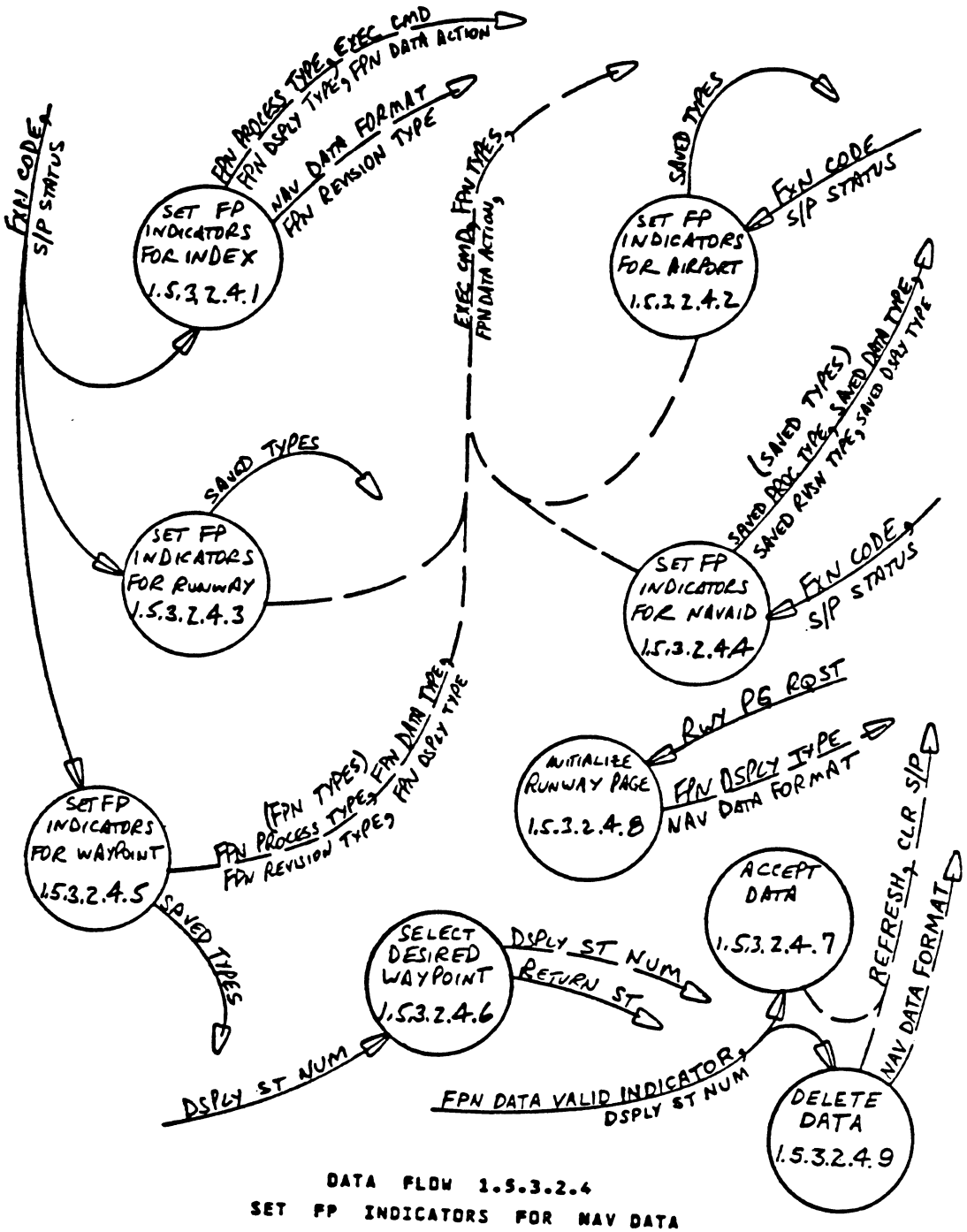


FIGURE 3 - Interconnections between DFD, CFD, and Control Spec.

4.3 Data Flow Diagrams.

DFDs are essentially identical to those in basic SA. The one exception is the appearance of data conditions (described earlier) flowing out of the primitive processes in which they are generated. They are shown there to complete the picture of the process, and are also shown flowing out of the same process on the corresponding CFD. Any further flow, to higher or lower levels, is shown only on the CFDs, as with all the other control signals. Figure 4 is a typical DFD with data conditions.

Process activators are not shown at all on the DFDs, only in the control specs. Since, in the document, a control spec is located close to its DFD, it is easy to refer to it to find which processes are activated. Processes which do not have activators operate in the same way as in basic SA - they are data triggered.



DATA FLOW 1.5.3.2.4
SET FP INDICATORS FOR NAV DATA

FIGURE 4 - Data Flow Diagram with Data Conditions

4.4 Control Flow Diagrams

The term "control flow diagram" is sometimes used synonymously with "state transition diagram", but this is not its meaning in this method. Here, the term is used because the diagram it describes is very similar to a DFD, so it is appropriate for them to have correspondingly similar names. State transition diagrams are referred to exclusively by that name in this method, and appear only in control specs.

Like DFDs, CFDs contain processes, signal flows, and stores, and must balance with their parent and child diagrams. The differences between the two are important, however, and are as follows:

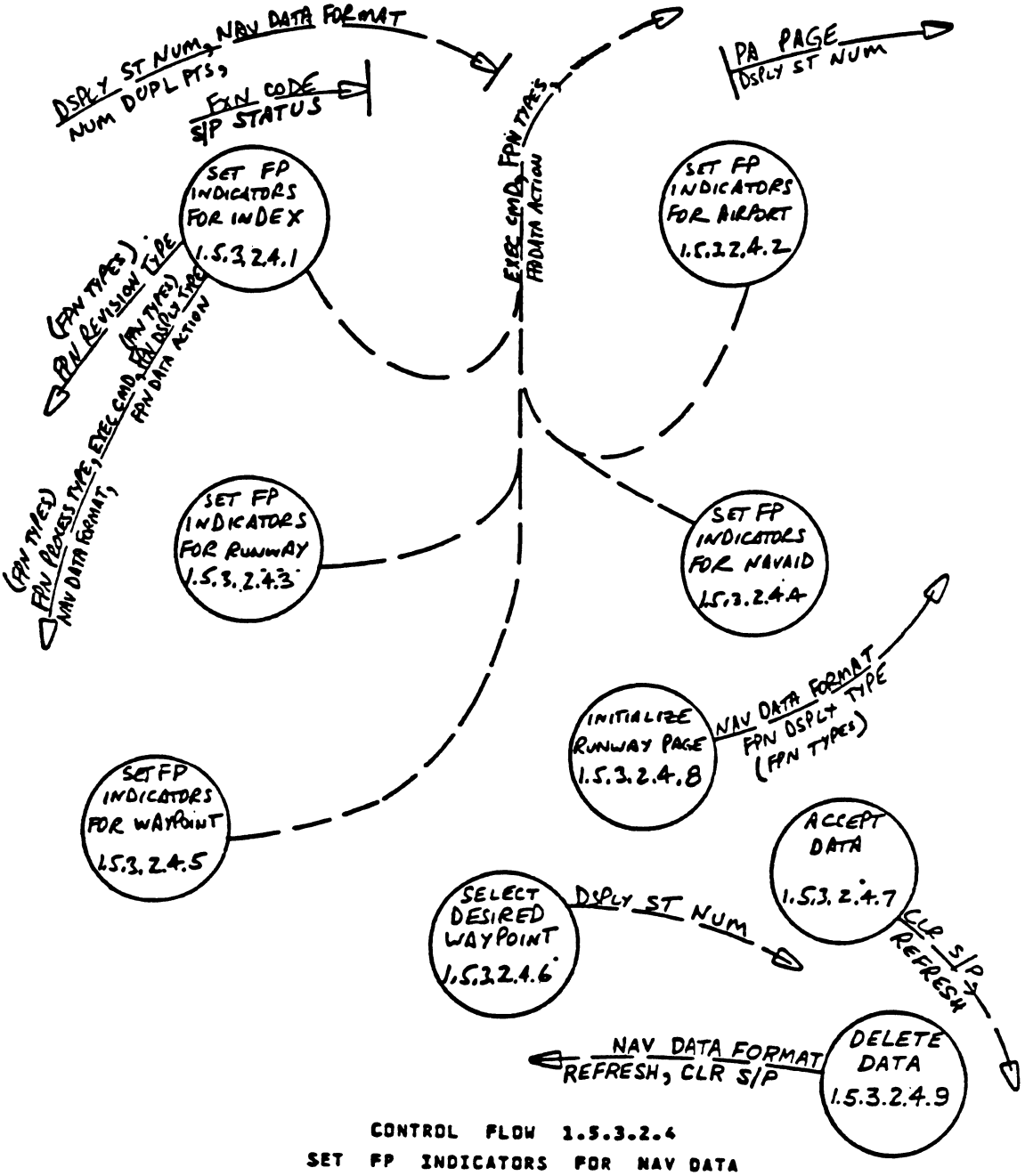
- their signal flows are control signal flows, and are shown with broken lines to distinguish them from data signal flows.
- signals flowing to and from the associated control spec are shown with a short bar on the end of the vector.
- the processes on a CFD are duplicates of those on the associated DFD. If a particular process on the DFD has no control signal flows associated with it, it may be omitted from the CFD.

It is not required that every DFD has a CFD and control spec associated with it. If none of the processes in the DFD is controlled, then a control spec is not required, and if none of the children of the DFD has any control signals associated it, then no CFD is required (no signals flowing down to or up from lower levels). However, if a control spec is needed, then so is a CFD (to provide the inputs and receive any outputs). Figure 5 is a typical CFD, and corresponds with the DFD of figure 4.

It is important not to misinterpret the control signals flowing into and out of processes on a CFD. They are not activators of those processes, but signals flowing between levels, just like data flows in DFDs. The process activators only appear in control specs.

4.5 Control Specs

Control specs contain the representations of the actual FS machines. Their purpose is analogous to that of process specs - to show how their outputs are generated from their inputs - but they do this using decision tables and state transition diagrams instead of structured English.



CONTROL FLOW 1.5.3.2.4
SET FP INDICATORS FOR NAV DATA

FIGURE 5 - Control Flow Diagram

FS machines may be divided into two types: combinational and sequential. In combinational machines, the current outputs and states of the internal elements are determined entirely by the current inputs. They are represented mathematically as a "3-tuple", {I, Z, w}, where:

I is a finite set of input symbols,
Z is a finite set of output symbols,
w is a mapping of I onto Z called the output function or transfer function.

Combinational machines are usually represented by decision tables in which all combinations of the input signal values (i.e. all the input symbols) are listed with their corresponding output signal values (output symbols). In practice, it is usual that many of the input symbols are of no interest ("don't care" condition) and the table can be greatly simplified. Figure 6 is a typical control spec using decision tables, including generation both of control signals and process controls. It corresponds with the DFD and CFD of figures 4 and 5. The numbers in the body of the process control table represent activation of the processes in that numerical sequence.

In sequential machines, current outputs and states of the internal elements are determined by current inputs together with past values of inputs and internal elements - i.e. they contain memory. They are represented mathematically as a "5-tuple", {I, Q, Z, d, w}, where:

I is a finite set of input symbols,
Q is a finite set of states,
Z is a finite set of output symbols,
d is a mapping of I X Q onto Q called the next state function,
w is a mapping of I X Q onto Z called the output function.

Sequential machines may be modelled in a number of ways, including the Moore model, in which the output function depends only on the current state, not on the inputs, and the Mealy model, in which the output function depends on both the current state and the inputs. It can be proved that any representation using one of these models has an equivalent representation on the other, but the Moore representation will usually require more states. Because of its greater flexibility, the Mealy model was chosen for this method.

Although the types of system we are dealing with are invariably sequential machines overall, when the control requirements are partitioned as described earlier, it is usually found that the sequential requirements can be concentrated into a few localized areas; that the rest of the control requirements can be represented in combinational machine form (simple decision tables); and that large parts of the system can be represented in basic SA form, with no control structure at all, using the "data triggering" concept.

CONTROL IN		CONTROL OUT	
DSPLY-ST-NUM	FXN-CODE	PA-PAGE	DSPLY_ST_NUM
5	6L	NONE	INIT/RET INDEX

CONTROL IN				PROCESS ACTIVATED 1.5.3.2.4.N			
DSPLY_ST_NUM	FXN_CODE	NAV_DATA_FORMAT	NUM_DUPL_PTS	1	8	6	7
5	1L, 2L, 1R	INDEX	>0	1	2	3	0
			0			0	3

CONTROL IN					PROCESS ACTIVATED 1.5.3.2.4.N						
DSPLY_ST_NUM	FXN_CODE	NAV_DATA_FORMAT	NUM_DUPL_PTS	S/P_STATUS	2	3	4	5	6	7	9
5	1L-5L, 1R-5R	AIRPORT	>0	D/C	1	0			2	0	0
			0	DELETE					0		2
				OTHERWISE						2	0
		WAYPOINT	>0	D/C	0			1	2	0	0
			0	DELETE					0		2
				OTHERWISE						2	0
		NAVAID	>0	D/C	0		1	0	2	0	0
			0	DELETE					0		2
				OTHERWISE						2	0
		RUNWAY	D/C	D/C	0	1	0			2	0

Sequential machines are represented using state transition diagrams, or various equivalent tabular forms, any of which may be used in a control spec. Figure 7 shows a typical example using a matrix form. In addition to the state transition diagram itself, decision tables are usually required to represent its input and output logic.

Since the control requirements for a given DFD may be arbitrarily complex, there is no restriction on the size of control specs, and they are frequently several pages long. It is important that their input and output signals are grouped together and clearly identified near the front of the spec so that they can be easily balanced with the CFD.

4.6 Requirements Dictionary

The requirements dictionary (RD), is essentially the same as the SA data dictionary, but it contains the definitions of both the data and control signals. The symbology is the same for both - control signals are grouped in just the same way as data signals. The RD has been automated using a commercial data management system, and is divided into fields: "name", "composed of", "used in", and "member of". The last two list, respectively, the flow diagrams in which the signal is used, and other signal groups in which it is included.

4.7 Timing Requirements

From the requirements point of view, timing falls into just two categories:

- required rates of receiving inputs and generating outputs
- response times from system input events to resulting system output events.

Input and output rates are stated in the requirements dictionary as attributes of the individual primitive signals. Response times are listed in simple tabular form showing the input signal(s), the event associated with those signals, the output signals, and the resulting event associated with those output signals. Figure 8 illustrates a response time spec format.

Such considerations as timing budgets for software functions or module calling rates have no place in a requirements spec.

STATE \ EVENT	SW TO CLB	TRNSTN TO CLB	TRNSTN TO CRZ	TRNSTN TO DES	PRED CMPLT INDIC	UPDT ACTV LATL GO TO MP	TODES MOVED	TODES MOVED WHILE STRTG DES	INIT CRNT ST	INIT DES
NO PLAN ENGAGED	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -
TAKEOFF	START CLB - - - - CLIMB	INIT VG ST - - - - CLIMB	- - - -	- - - -	PREPARE FOR HOLD - - - -	CLR TODES DIST CORR, LATER MSG - - - -	PREPARE FOR DES		- - - -	- - - -
CLIMB	- - - -	- - - -	LATER MSG - - - - CRUISE	- - - -	- - - -	TEST CLB HLD, UPDATE TODES - - - -		PREPARE FOR DES, INIT VG ST	PREPARE FOR CRZ PREPARE FOR DES PREPARE FOR HLD - - - -	PREPARE FOR DES, INIT VG ST
CRUISE	- - - -	- - - -	- - - -	DLT DES MSGS - - - -	- - - -	TEST CRZ HLD, UPDATE TODES			SET CRZ PREPARE FOR DES PREPARE FOR HLD - - - -	- - - - PATH DESCENT
EARLY DESCENT	- - - -	- - - -	- - - -	- - - - PATH DES	- - - -	- - - -		- - - -	TST DES MODE, SET CRZ PREPARE FOR DES PREPARE FOR HLD - - - -	TST DES MODE, PREPARE FOR DES INIT VG ST - - - - PATH DESCENT
CRUISE CLIMB	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	PATH DESCENT	SET CRZ PREPARE FOR DES PREPARE FOR HLD - - - -	PREPARE FOR DES, INIT VG ST - - - - PATH DESCENT
AIRMASS DESCENT	- - - -	- - - -	- - - -	- - - -	- - - -	TEST AIRM DES HOLD, UPDATE TODES - - - -			SET DES PREPARE FOR DES PREPARE FOR HLD - - - -	SET DES PREPARE FOR DES, INIT VG ST - - - - PATH DESCENT
PATH DESCENT	- - - -	- - - -	- - - -	- - - -	- - - -	TEST PATH DES HOLD, UPDATE TODES - - - -		PREPARE FOR DES, INIT VG ST - - - -	- - - -	- - - -
LATERAL PATH ONLY	- - - -	- - - -	- - - -	- - - -	- - - -	TEST LATL HOLD	- - - -	- - - -	- - - -	- - - -

Control Spec 2.5 : Generate Vertical Guidance.
Sheet 9 of 19.

ACTIONS	blank => do nothing
- - - - NEXT STATE	blank => stay in current state

FIGURE 7 - Part of Control Spec using State Transition Matrix.

INPUT SIGNAL	EVENT	OUTPUT SIGNAL	EVENT	MAX RESPONSE TIME
A/N_KSTRK	ANY CHARACTER	SCRTCHPD	CHARACTER APPEARS	1/3 SEC

FIGURE 8 - Response Time Spec Format

5.0 Preparing Specs using the Real-Time SA Method.

The guidelines for preparing basic SA specs generally apply to real-time SA specs. In addition, some further guidelines have been found useful, as follows:

- The customer spec usually has data and control requirements totally intermixed, so it is necessary to start separating them before starting the analysis. This can be a long and tedious task and, in fact, tends to continue throughout the analysis.

- Work on the data flow structure first, or at least start it first. This follows the principle of defining what has to be controlled before deciding how to control it.

- Try to minimize the amount of control specified. In other words, maximize the amount of basic SA in the spec. Control tends to be implementation dependent and the maximum possible freedom should be given to the designer.

- Try to keep as much of the control requirements as possible in combinational machine form (decision tables). This is the simplest and therefore the most desirable form.

- Concentrate the requirements which must be in sequential form into localized areas.

- Try to put the control requirements at as high a level as possible in the structure. Control specs typically transform into "boss" modules in the structured design, and these decision making processes should be towards the top of the hierarchy.

6.0 Practical Experiences with the Method.

Everyone with experience in the use of structured methods recommends that new users introduce them on a small, low-key project. The project on which this method was introduced was very large (the SA spec is 15 volumes long) and very critical, and moreover, the method itself was new and untried. To counter these obstacles, all the training was done in-house, and the methods team acted as full time consultants to the project staff, providing advice, assistance, and practical problem solving on demand. The level of acceptance varied widely from individual to individual, and there were some difficulties in getting consistent standards from all of the 20 to 25 engineers working on the requirements definition, but the majority were overwhelmingly positive towards it, and the results have generally been excellent. FMCS is in system test at the time of writing and is performing significantly better at this stage of development than previous, similar systems.

Customer acceptance, too, has been excellent, and they are using this method as a model for their own work and as a standard for their other vendors.

Another, and possibly the most important, advantage we had was that, after the decision was made to proceed with structured methods, there was 100% management commitment behind the effort. There would have been no time to deal with political problems which others apparently have had to contend with, and, happily, none occurred.

As expected, the "up front" effort to prepare the requirements spec was considerably more than on previous projects, in fact, considerably more than was originally estimated for this project. Nevertheless, the project overall is on schedule, and the results of the additional effort in terms of performance to date, and improved communication with the customer and with the design group, justify this expense.

The most serious shortcoming has to do with the size of the system rather than the method itself: manual implementation is impractical on systems of this size, and full automation is essential in the long term.

7.0 Summary.

A real-time structured analysis method has been developed in response to the needs of a large, complex avionics system. It is based on the Yourdon SA method, but adds two new features: control flow diagrams, and control specs. The former are essentially the same as data flow diagrams but show flow of control signals, and use a special symbol to show flow to and from control specs; the latter show the actual control requirements of the system, and make use of well known techniques from finite state machine theory. The new features have been integrated completely into the existing SA structure, using the same leveling, balancing, and numbering techniques. The control structure is slaved to the data structure, so that the controls for a given DFD are concentrated close to that DFD. The method has been successfully applied to the system for which it was originally developed, and will be used on our future development programs. In addition, considerable interest has been expressed by other organizations involved in the development of real-time systems.

8.0 Acknowledgments.

I would like to thank Lear Siegler Inc., Instrument Division, for permission to publish this article, and K. Hornbach, D. Morrow, W. Roth, Dr. R. Wierenga, and G. Wood for their review and valuable comments.

DECUS SUBSCRIPTION SERVICE
DIGITAL EQUIPMENT COMPUTER USERS SOCIETY
ONE IRON WAY, MRO2-1/C11
MARLBORO, MASSACHUSETTS 01752

Bulk Rate
U.S. Postage
PAID
Fitchburg, MA
Permit No.
21

MOVING OR REPLACING A DELEGATE?

Please notify us immediately to guarantee continuing receipt of DECUS literature. Allow up to six weeks for change to take effect.

( ) Change of Address
( ) Delegate Replacement

DECUS Membership No.: _____

Name: _____

Company: _____

Address: _____

_____

State/Country: _____

Zip/Postal Code: _____

Mail to: DECUS - ATT: Subscription Service
One Iron Way, MRO2-1/C11
Marlboro, Massachusetts 01752 USA