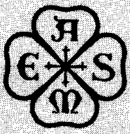

THE AMERICAN SOCIETY OF MECHANICAL ENGINEERS
345 E. 47 St., New York, N.Y. 10017

82-GT-251

The Society shall not be responsible for statements or opinions advanced in papers or in discussion at meetings of the Society or of its Divisions or Sections, or printed in its publications. Discussion is printed only if the paper is published in an ASME Journal. Released for general publication upon presentation. Full credit should be given to ASME, the Technical Division, and the author(s). Papers are available from ASME for nine months after the meeting.
Printed in USA.

Copyright © 1982 by ASME

An Approach to Software for High Integrity Applications

W. C. Dolman
Assistant Performance Manager

J. P. Parkes
Performance Project Engineer

Lucas Aerospace Limited,
York Road,
Birmingham, England

This paper outlines one approach taken in designing a software system for the production of high quality software for use in gas turbine control applications. Central to the approach is a special control language with its inherent features of visibility, reliability and testability, leading to a software system which can be applied to applications in which the integrity of the units is of prime importance. The structure of the language is described together with the method of application in the field of aircraft gas turbine control. The provision of documentation automatically is an integral part of the system together with the testing procedures and test documentation. A description of how these features are combined into the total software system is also given.

INTRODUCTION

This paper describes the software system designed and developed by the Engine Electronics group of Lucas Aerospace Limited to provide high integrity software for its digital control units.

OBJECTIVES OF A HIGH INTEGRITY SOFTWARE SYSTEM

No matter how well designed the hardware of a microprocessor based control system may be, the success of the project hinges on the quality of the software used. Quality, in this context, encompassing;

simplicity of application, visibility, controllability and modification capability.

Control Language Concepts

However the control program is written, it needs to be converted into a series of instructions capable of being "understood" by the particular microprocessor in use. This series of instructions, known as the runnable program, is held in the store of the microprocessor system in binary form.

The diagram, Fig 1, shows the various means by which the source program can be converted into a runnable program. The simplest of these is where the programmer produces the program directly in the binary form required by the microprocessor. Although effective and efficient programs can be produced in this way, the difficulty in meeting the Quality requirements outlined above prevents the method from being viable, especially if the software is to be of a standard for use in a civil aircraft application.

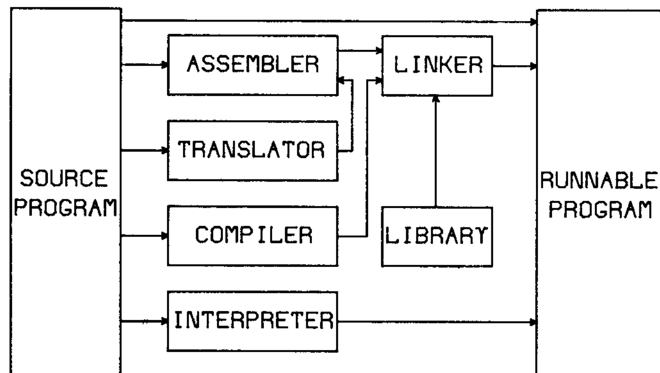

FIG 1. SOURCE PROGRAM CONVERSION METHODS

Contributed by the Gas Turbine Division of the ASME.

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

Some of the disadvantages of the direct approach can be mitigated by writing the programs in assembler language and this method has been used, successfully, in some of our earlier projects. Some quite serious disadvantages relating to the quality of the software, however, still remain.

High level language, where the source program is written in a form near to normal English, allows most of the disadvantages of the previous approaches to be reduced to an acceptable level. These high level languages can be converted into a runnable program by one of three approaches, by means of a translator, a compiler or an interpreter. The translator approach confers most advantages for the type of application being considered, because it removes from the software approval loop the use of complex compilers which are intrinsically error prone.

When considering the needs of a suitable high level language for control system applications, six objectives of such a language were identified, these being:

1. The language to be, as far as the control engineer is concerned, independant of the microprocessor being used.

2. The control engineer to be relieved of the need to have an extensive knowledge of the microprocessor being used.

3. The user interface to be tailored to produce an application-oriented programming system so that the control engineer can transcribe his control system design into runnable software accurately and rapidly.

4. The software stages to have full visibility so that traceability of the software from the original control system to the runnable code in the system store is ensured.

5. The software system to be so structured that good design practices are enforced and that rigorous testing and cross checking can be applied.

6. The software system to be structured so that the raising of high quality documentation becomes a natural and integral part of the software writing process.

With these six objectives in mind, together with a knowledge of the technical needs of the control engineer, the Engine Electronics Group of Lucas Aerospace developed a high level language known as LUCOL[1] (LUcas COntrol Language) as part of a total software system package. The language enables control engineers to program directly from their control diagrams, so that they and other engineers can have full software visibility in a form traditional to engineering disciplines.

LUCOL

The basis of the system is a series of modules representing commonly used analogue type control system blocks as well as sequential logic operations. The control engineer solves his problem by specifying an appropriately ordered network of modules. These modules are drawn from a library of rigorously tested assembler language programs which also includes input-output and safety routines.

Each module has assigned to it a mnemonic identifier and a standard functional diagram. The control engineer draws his system block diagram using the LUCOL elements - this block diagram then forms a pictorial representation of the software.

The basic control source program is generated simply by producing a calling sequence listing the modules, in mnemonic form, and their associated parameters. These parameters specify 1. the data flow between modules (analogous to the signal flow on a conventional block diagram) and 2. the direct parameters such as gains, time constants etc.

A feature of LUCOL is that a module may use the output of the previous module as an input automatically. This method of transference, termed "implicit flow" is employed by those modules which are closely coupled and results in an improvement in the efficiency and clarity of the resultant control program. Explicit flow, where the input and/or output is defined in the calling sequence, is used for example, with input modules where several parallel operations are likely. This, again, is to optimise overall efficiency.

LUCOL LANGUAGE AND CONCEPTS

The LUCOL system aims to provide an integrated approach to the production of high quality software. The LUCOL language provides the centrepiece of the system around which software production, testing and documentation facilities have evolved.

Modularity

LUCOL program code is strictly modular. Thus expressions of the form:

```
(1)  A = SQRT (B * C/D),
```

are deliberately avoided. The LUCOL approach is:

```
(2)  MULDIV (B,C,D)  ;  Fetch B into implicit
                        register
                     ;  Multiply by C, Divide
                        by D
     ROOT            ;  Take square root of
                        implicit register
                        contents
     PUT    (A)      ;  Store in location A
```

The convenience of being able to perform multiple operations via a single expression is therefore foregone, but two significant advantages outweigh the tedium of explicit coding. Construct (1) requires a bug-free parser (which is impossible to guarantee, and even if the translator/compiler enjoys a large amount of user confidence, non-users will not share such convictions), and requires close inspection of the code generated from each compound expression. Construct (2) on the other hand lends itself to direct and immediately visible translation into a series of calls with parameters to the modules MULDIV etc. These modules are termed LUCOL MODULES and the rigorous testing procedure of the LUCOL modules ensures that such a strictly modular program as LUCOL is founded only upon solid, reliable building blocks.

---
[1] LUCOL is a registered trade mark of Lucas Aerospace Limited.

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

Single Accumulator Concept

As may be inferred from the above example, a LUCOL program appears as if coded for a single accumulator machine. The accumulator is referred to as the "implicit register". This feature was incorporated for reasons of efficiency and to provide a natural signal flow through the program. The basic modules used to load and store the implicit register are GET and PUT, and most other modules operate upon the implicit with ancillary inputs and outputs. In addition a certain class of modules will perform a GET prior to performing any computations. Thus for instance:

```
MULDIV (B, C, D)
```

fetches the contents of location B into the implicit register prior to performing the multiplication and division. In all such cases it is permissible to enter an asterisk in place of the appropriate parameter which causes the preliminary fetch to be omitted:

```
MULDIV (*, C, D) ; Multiply contents of Implicit
                   Register
                 ; by C/D
```

An asterisk is used throughout to indicate the current contents of the implicit register.

To illustrate the above procedure, take the requirements defined by the system diagram in Fig.2.

This shows a proportional plus integral control loop for variable T6 which, when passed through a lowest wins with a function of P3, generates a fuel demand signal. The first step is then to convert this to a LUCOL flow chart as shown in Fig.3.

As can be seen, Fig.3 is easy to compare with Fig.2 and does not require extensive training to understand. From the LUCOL flowchart, the corresponding listing is produced (Fig.4). This is done simply by listing the module mnemonics and adding the parametric information. Comments may be added by using a semicolon separator. The listing for Fig.3 would be:-

```
GET    (T6D)          ;Read T6D INTO IMPLICIT REGISTER
SUB    (T6A)          ;SUBTRACT T6A TO FORM ERROR SIGNAL
PUT    (T6E)          ;SAVE ERROR SIGNAL
MULDIV (*,60,100)     ;SCALE ERROR BY 0.6
INTR   (GAIN,STORE)   ;INTEGRATE RESULT
PUT    (INTGRL)       ;SAVE RESULT
MULDIV (T6E,20,100)   ;SCALE T6E BY 0.2
ADD    (INTGRL)       ;ADD INTEGRAL TERM
PUT    (T6LOW)        ;SAVE RESULT
GET    (P3)           ;READ P3 INTO IMPLICIT REGISTER
FGEN1  (TABLE)        ;LOOK UP P3 SCHEDULE
LWINS  (T6LOW)        ;LOWEST WINS WITH T6 LOOP ERROR
PUT    (FUEL)         ;SAVE FUEL DEMAND
```

SECTION OF A CONTROL DIAGRAM
PROPORTIONAL PLUS INTEGRAL CONTROL LOOP

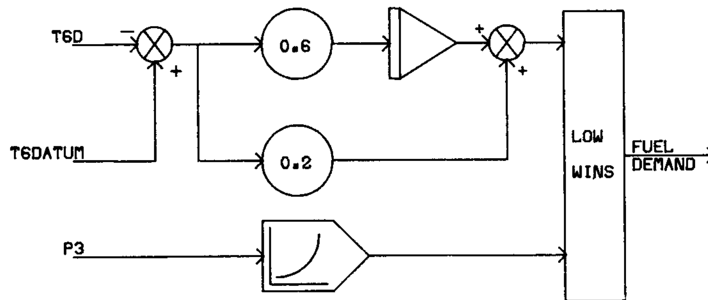

FIG 2. CONTROL SYSTEM EXAMPLE

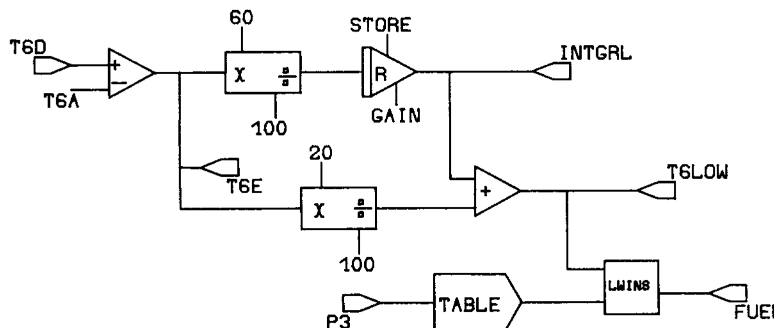

FIG 3. TYPICAL LUCOL FLOW CHART

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

LUCOL Program Structure

The system is designed so that a program can be run with a minimal executive. Thus all aspects of program execution may be inferred from the LUCOL source code without foreknowledge of the scheduling algorithms, input/output queues and other appurtenances normally associated with an executive. In any event, such executives are impossible to debug and test fully as is evidenced by the number of patches supplied to users of such systems for some considerable time after the system becomes commercially available. The LUCOL approach is again simplistic involving direct connection to interrupts and self-linking code.

A LUCOL program is divided into logical sub-units known as segments:

```
SEGMENT CONTRL      ;DECLARE A SEGMENT NAMED
    :                "CONTRL"
LUCOL MODULE CALLS ;
    :
ENDSEG              ;DECLARE END OF SEGMENT
```

Segments are grouped into "Levels" numbered from zero upwards. Level 0 corresponds to the base level program which executes cyclically whilst no interrupts are being serviced. Higher numbered levels correspond to the various interrupts to be serviced on a particular hardware configuration. Level declarations consist of two statements: a level and a call to either of the modules EXEC or PWRUP. PWRUP is specific to Level 0 whilst EXEC must be used at all other levels. A typical declaration of the levels to be serviced might be:

```
LEVEL 0               ;INITIALISATION AND BASE
                       LEVEL SEGMENTS
PWRUP (INIT,BASE)
LEVEL 8
EXEC (WDT)            ;SERVICE WATCHDOG TIMER
LEVEL 14              ;SAMPLE RATE CLOCK INTERRUPT
EXEC (INOUT,VALID,
      CNTRL,SAFETY);CONTROL SEGMENTS
```

At level 0 the first segment in the call to PWRUP is the "initialisation segment" which is executed whenever a power up/reset occurs. Subsequent segments make up the base level program. At higher levels the priority of interrupt decreases with increasing level number. The segment WDT which services the watchdog timer executes at a higher priority than the control. The control segments (Input/output, Data validation, Control calculations and safety) execute sequentially in response to an interrupt from the sample rate clock.

Program Flow

Logical decisions are incorporated in a LUCOL program via IF-THEN-ELSE and CASE constructs.

```
DOIF (FLAG)
   :                  ;EXECUTE THIS SECTION IF
   :                   "FLAG" IS TRUE
   :
ELSE
   :                  ;THIS SECTION EXECUTED IF
   :                   "FLAG" IS FALSE
ENDIF
   :
CASE (INDEX,CASE1,
      CASE2,CASE3)
BEGIN (CASE1)         ;EXECUTE IF INDEX = Ø
   :
BEGIN (CASE2)         ;EXECUTE IF INDEX = 1
   :
   :
BEGIN (CASE3)         ;EXECUTE IF INDEX = 2
   :
   :
OTHERWISE             ;EXECUTE IF INDEX<0 OR INDEX>2
   :
   :
ENDCASE
```

LUCOL procedures may be invoked via the CALL module; and may be called as many times as required from any segment.

```
CALL (SUBA)
```

A procedure differs from a segment only in that it is preceded by a PROCEDURE declaration and terminated by a RETURN statement:

```
PROCEDURE SUBA        ;DECLARE PROCEDURE "SUBA"
    :
    :
LUCOL Module calls
    :
    :
RETURN                ;RETURN FROM PROCEDURE
```

The WHILE-ENDWHILE and GOTO statements are considered too dangerous to incorporate in flight-standard software and are not recognised by the LUCOL translator. Both may lead to infinite loops, whilst the arguments against GOTO and in favour of block-structured logical flow are well rehearsed.

LUCOL Data Types

All data storaged used by a LUCOL program must be declared as either constant, variable, logical or text. Constants and text occupy read-only memory whilst variables and logicals occupy read/write memory, the latter taking only the two values TRUE and FALSE. Example data declarations are:

```
VARIABLE VARI       ;RESERVES A  1 WORD VARIABLE
VARIABLE VAR2 [10] ;RESERVES A 10 WORD VARIABLE
LOGICAL  FLAG1      ;RESERVES A  1 WORD VARIABLE
VARIABLE VA3
 (VAR1, VAR2)       ;RESERVES 11 WORDS
                    ;1st IS ADDRESSABLE AS VAR1,
                    :SUBSEQUENT TABLE OF 10
                    :ELEMENTS AS VAR2
CONSTANT C1(999)    ;1 WORD CONSTANT CONTAINING
                     DECIMAL 999
CONSTANT C2(>FFFF,Ø);2 WORD CONSTANT CONTAINING
                    :HEXADECIMAL FFFF AND Ø
```

Translation

The essential function of the LUCOL translator (that of translation) is simple. Thus the LUCOL module call:

```
MULDIV (B, C, D)
```

translates to four words of assembler code, for example in Macro assembly notation for the Digital Equipment Corporation's PDP[2]11 series of computers,

```
.WORD MULDIV, B, C, D
```

[2] PDP is a trademark of DEC.

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

Subsidiary but vital functions are error detection and the creation of internal symbol tables and bit maps to interface with the LUCOL symbolic debugging aids.

Primary error detection involves the inspection of module calls to ensure that:

(1) The module is recognised by the LUCOL system.

(ii) All named parameters have been defined within the program.

(iii) Each parameter is of the appropriate type (variable, logical, constant, etc).

(iv) The module is called with the correct number of parameters.

Input to the translator consists of a definitions file and any number of segment/procedure files. The definitions file contains all Level and storage declarations. Symbols defined by the user are global to all segments and procedures appearing in subsequent files. Input files are grouped together via filename in the primary input known as the "Group File".

Output from the translator consists of a program listing and a threaded-code assembler file. An optional symbolic cross-reference of the LUCOL program may be appended to the listing.

LUCOL MODULES

The function of a module must be sufficiently simple to enable it to be tested automatically by setting inputs and reading outputs without resorting to breakpoints or checking intermediate calculations. Modules are written in the native assembler language of the target microprocessor, and are cross-assembled to form the object libraries with which the translated LUCOL program is linked. The cross-assembler which was designed to be easily re-configured to support different microprocessors was also designed and written by the Engine Electronics Group of Lucas Aerospace. As already stated, modules are self-linking. A register is reserved for threading the LUCOL code. At entry to each module this register is pointing to the address of the first of that module's parameters. The module accesses its parameters via this register (preferably via auto-increment if available) and exits via a jump to the following module whose first parameter should now be pointed at by the register.

A further requirement is that modules be re-entrant. Although a module may be called from various levels and therefore from different interrupts, only one copy of that module is linked in with the LUCOL program. Re-entrancy is assured by avoiding the use of any scatch-pad memory within a module and checked when the module is tested. If a module needs to maintain, for instance, historic values (e.g. an integrator), then a read/write memory location for that historic value is provided explicitly in the module call. The modules are fully tested automatically using a program termed LMTP (LUCOL Module Test Program) which accepts as input a test schedule and outputs a test report on the results of the testing. This report together with the module source listing forms a comprehensive documentation package for the modules.

DOCUMENTATION

Flight-standard software is bound to impose a heavy requirement on the supplier for documentation. Much effort has been expended to produce documentation automatically and so reduce the work-load on the control designer.

An integral part of the LUCOL language is an "automatic" documentation system. This system will, by automatically scanning the inter-connected source files, produce a unique build standard for the LUCOL program.

Certain data are required by the system, e.g. author's name, date, project name, functional description, input-output requirements, modification history, etc. These data, together with the LUCOL list, are then automatically compiled into a report defining the software standard of the control system. This automatic report generating system ensures that all relevant data are provided in a standard format and that a complete record of the software standard of a control system and its modification status are maintained.

Timings of LUCOL programs by Level and Segment down to contributions by individual modules are produced by program. Data structures such as carpets and function generation data are verified, again automatically.

The most onerous burden that has so far been removed from the designer is the production of the LUCOL diagrams (flow charts).

LUCOL Diagrams

The overall aims of the LUCOL Diagram are:-

1. To document the software in a visible format which is understandable by everyone involved in a project whether familiar with LUCOL or not.

2. To provide a link between control diagram and the detailed software.

3. To provide information on both signal and program flow.

General Format. Within these aims the most difficult problem to solve is visibility since the link between the written specification and the program code - can be biased towards either signal or program flow, the choice being mainly subjective.

In practice most of the decisions hinge on whether signal flow or program flow is chosen as the major feature of the diagrams. In either case the requirements of the other can be catered for by means of identifiers, i.e. all signals labelled on the program flow diagram; order of execution labelled on the signal flow diagram.

For LUCOL Diagrams the normal convention is to show signal flow as the primary parameter for the following principal reasons:

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

1. It is required to be written in advance of the program by people who, although probably familiar with software are not required to be expert programmers.

2. The signal flow is relatively well defined in advance of the program flow, enabling the LUCOL Diagram to be written at the earliest stage, with less need for continuous updating.

3. A signal flow diagram is a more suitable vehicle for the detailed specification of modifications by control engineers and other non-specialist programmers associated with a project.

4. Program flow charts can readily be produced retrospectively if required for documentation purposes. This could also be performed automatically.

Wherever possible the following conventions when drawing LUCOL diagrams should be applied.

(i) Every connection on the left of a module diagram is a signal entering.

(ii) Every connection on the right of a module diagram is a signal leaving.

(iii) Every connection on the top or bottom of a module connection is a stored constant or variable used by the module.

If the above conventions cannot be applied, then deviations from the conventions must be explicitly stated by the judicial use of arrows.

LUCOL Plotter Interface

LUPIN (LUcol Plotter INterface) is a program which will accept as input a LUCOL segment or procedure file and produce as output a diagram of the segment procedure on a drum plotter without further intervention by the operator.

An example of the output is shown in Fig.4, and the output for the example used in Figs. 2 and 3 is shown in Fig.5. Figs. 4 and 5 are on the following pages.

DEBUGGING AIDS

Three symbolic debugging aids are available: the Multi-Module Test program (MMTP), the Full Change Program (FCP) and the On-line Display Program (ODP).

MMTP

This is an emulator which runs on either the host computer system or the target microprocessor development system. Breakpoints may be set, variables inspected and modified and individual instruction steps within modules may be traced. The LUCOL interrupt structure is catered for in that an interrupt to a higher priority level may be set to occur anywhere within the program. The program does not run in real time however and does not interfere with the control hardware.

FCP

The FCP monitor is linked in with the LUCOL program. The ROM's produced are inserted in a RAM-based development system and downloaded into RAM. FCP provides the ability to monitor and modify LUCOL programs. Modifications are implemented via patches to the RAM copy of the program. To make successful patches permanent the engineer must return to his LUCOL program source code, make the necessary edits and rebuild. This provides an essential check on the chaos likely to ensue from a succession of undocumented patches. Since the system has the ability to make quite significant changes to the control program, it is used with engine simulation rather than on live engine tests. The LUCOL program runs in real time.

ODP

This program resides in a monitor box which is independant from the control which is to be tested. A full symbol table of the control is made available via a plug-in ROM. Communication with the unit under test is established via a test highway. ODP is an attenuated version of FCP in that it allows the control to be run and monitored, but has no facilities for making patches to the LUCOL program. It has the advantage that it may be used to monitor the operation of a control program, whilst the control is running on an engine test, without allowing the possibility of an inadvertent change being made to the program.

PDP 11/70 SYSTEM

One of the major benefits of the system so far described is that it runs completely on the PDP 11/70 Engineering Computer System. This means we can go from the LUCOL source program right through to the prommable code and then into the actual proms without the use of any microprocessor software development. This gives us a great advantage when implementing another microprocessor as we do not necessarily require to purchase the relevant microprocessor software development system or need to learn a new operating system associated with the equipment. A brief block diagram of the PDP 11/70 system is shown in Fig. 6. The broken lines indicate the areas where documentation is automatically produced.

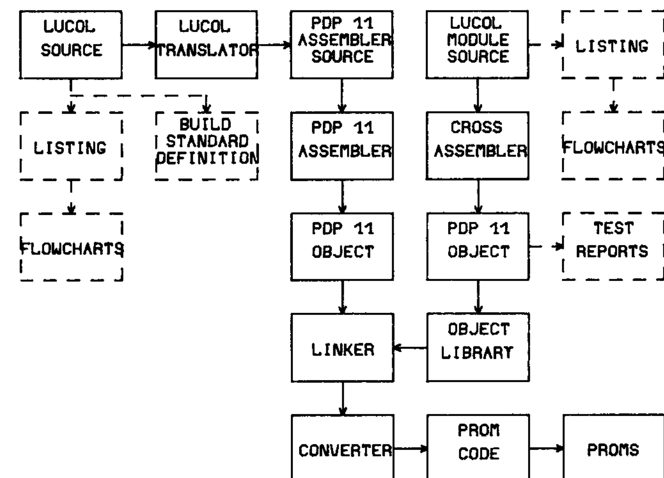

Fig. 6 PDP 11/70 LUCOL Software System

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

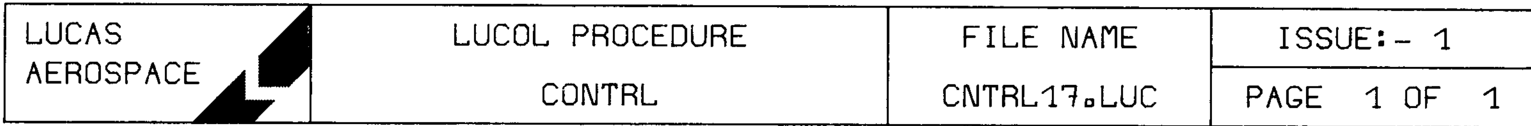

Fig. 4 An example of output from the LUCOL Plotter interface.

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

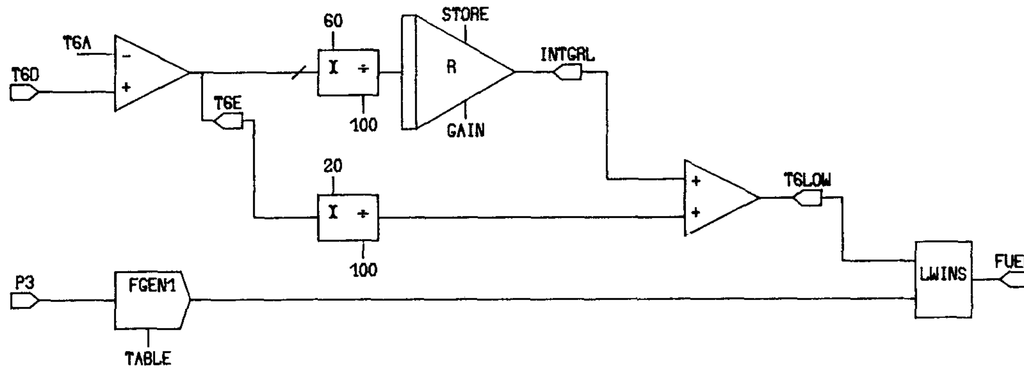

Fig. 5 LUCOL Plotter interface output for Figs. 2 and 3.

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

As the 11/70 is a multi-user multi-tasking system many individual users can create and develop LUCOL control programs at the same time, giving a distinct advantage over using microprocessor software development systems.

BUILD STANDARD CONTROL

The build standard of a particular control software is automatically governed by the issue numbers of the individual constituent software modules that are linked together to form the runnable program. The included constituent modules, both LUCOL modules and segments, have their issue number embedded in their file name so that the different issues of the same modules and segments are in fact held in completely different files on the PDP 11/70 discs. The build standard of a particular project LUCOL module library is defined as consisting of various LUCOL modules each of a particular issue, the library name itself having the library issue embedded in its name. If the library has a LUCOL module deleted from it, added to it, or replaced by another issue of the same LUCOL module then the library issue itself is updated, The instructions for amendment of the library are issued and controlled by the well known and approved company procedure for build standard amendment of hardware and delivered units. The input to the LUCOL translator is what is known as a 'GROUP' file which itself is controlled by an issue number embedded in its name and the name is in fact the identity given to the control software built from the group file. The group file contains the file names of the LUCOL segments and procedures, consequently their issue number, that make up that particular issue of the build standard. The contents of the group file, segments and procedures are strictly controlled by the same method as that described for the LUCOL module library.

The method of implementing an approved change is strictly controlled and documented as part of the company's Quality procedures, to make sure that the appropriate documentation changes are made to ensure as much as possible maximum visibility.

SOFTWARE QUALITY CONTROL

An integral part of the LUCOL system, as already stated, is an automatic documentation system. This system will, by automatically scanning the interconnected source files, produce a unique build standard for the LUCOL program.

Certain data are required by the system, e.g. author's name, date, project name, functional description, input/output requirements, modification history, etc. These data, together with the LUCOL list, are then automatically compiled into a report defining the software standard of the control system. This automatic report generating system ensures that all relevant data are provided in a standard format and that a complete report of the software standard of a control system and its modification status are maintained.

Before a module can be introduced into the LUCOL library or an existing module modified, a defined sequence of operations have to be carried out:-

1. The control engineer recognises a requirement for a new module.

2. A definition of the functional requirements is generated together with a test procedure.

3. The functional requirements and test procedure are examined to ensure that:-

   a) the inclusion of the module into the library is justified; i.e. the requirements cannot be met by an existing module or by a combination of existing modules.

   b) the performance and test requirements are adequately defined.

4. Detail software for the module is then written and tested and the resultant documentation checked by means of a software technical committee. The software technical committee consists of suitably qualified engineers drawn from the various engineering departments involved in digital design.

5. The documentation package is then submitted to the GEM for formal approval. The GEM (General Engineering Meeting) has the responsibility to control and maintain the company's engineering standards of design.

6. Having received approval, the module is issued for use.

This procedure ensures that the modules available for use are fully defined, specified and tested and that the information relating to these is presented to the user in a consistent form. The documentation package for a module includes:

1. Definition of the module function and application including the symbolic and mnemonic representations to be used.

2. Test specification.

3. Program listing.

4. Test results.

The LUCOL lists defining a given control system are subject to a procedure similar to that shown above for modules. Again, formal approval at the GEM is required before the control software can be issued. Any modifications carried out in the field, i.e. during engine tests, have to be incorporated into the standard of software previously agreed and resubmitted to the GEM for approval. Note that modifications to the software relating to a specific module cannot be carried out in the field.

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use

CONCLUSIONS

This paper has attempted to describe a flexible software system for the design and development of control software for use in high integrity applications. Flexible in providing control engineers with a high level language and debugging aids, allowing the creation of software to be less time consuming than traditional methods. Combined with this flexibility are the automatic restrictions and documentation procedures embedded in the system to produce software of high integrity and engineering visibility.

ACKNOWLEDGMENTS

The authors would like to thank the Directors of Lucas Aerospace Limited for permission to publish this paper. They would also like to acknowledge the skill and hard work contributed by their colleagues, both in the U.S.A. initially and England, in the design and development of the software system described in this paper.

Downloaded From: https://proceedings.asmedigitalcollection.asme.org/ on 01/24/2016 Terms of Use: http://www.asme.org/about-asme/terms-of-use