UDC 681. 3.02

Computer-Aided Program Analysis System (CAPAS)

● Hirotoshi Tono ● Minoru Takahashi ● Hisanori Yasugi

● Hiromi Uchimaru

The increased use of microcomputers in electronic controllers for automobiles has caused an increase in control program design work. Such factors as pollution regulations have made more comprehensive control mechanism necessary, and attached greater importance to safety. These, in turn, have contributed to the program designer's work load.

The Computer-Aided Program Analysis System (CAPAS) we developed analyzes instructions in a program designed in assembler, and presents them in an easy-to-understand form. Unlike a flowchart generator, CAPAS determines the branching conditions and data to be stored in memory, and generates flowcharts and their explanations. CAPAS enables designers with little experience to understand program functions. It also enables programs to be reused or developed by more than one designer without causing that part of the program for which a designer is not responsible to become a black box. CAPAS is expected to reduce debugging time and improve program quality. This report presents the basic principles of CAPAS and tells how it is applied to program module design.

1. Introduction

The application of microcomputers to electronic automobile control began in the second half of the 1970s with their use in engine control as part of emission regulation. Currenty, microcomputers are used in a number of automobile components including those related directly to automobile performance such as transmissions, suspensions, and brakes and those contributing to user comfort such as air conditioners, audio systems, and navigation tools. Microcomputers implement complicated control algorithms. They are being applied to an increasing number of fields. Manuafacturers who must handle complicated large-scale control programs are short of staff, however, to develop programs.

Control programs, usually of processing capability and memory capacity, are generally written in assembler. The introduction of higher performance microcomputers and of larger capacity, lower price memory has made it possible to design programs in languages such as C. Engine control requires memory with relatively low capacity to process many I/O signals at high speed, so it is difficult to go completely to advanced languages. Thus, assembler continues to play an important role.

Major problems of program design in assembler consist in its incomprehensible character and many debugging measures. This paper describes problems entailing more efficient program development for automobile control. It also introduces a system, CAPAS, which provides program development support by analyzing and displaying how programs written in assembler run.

2. Problems entailing efficient program development

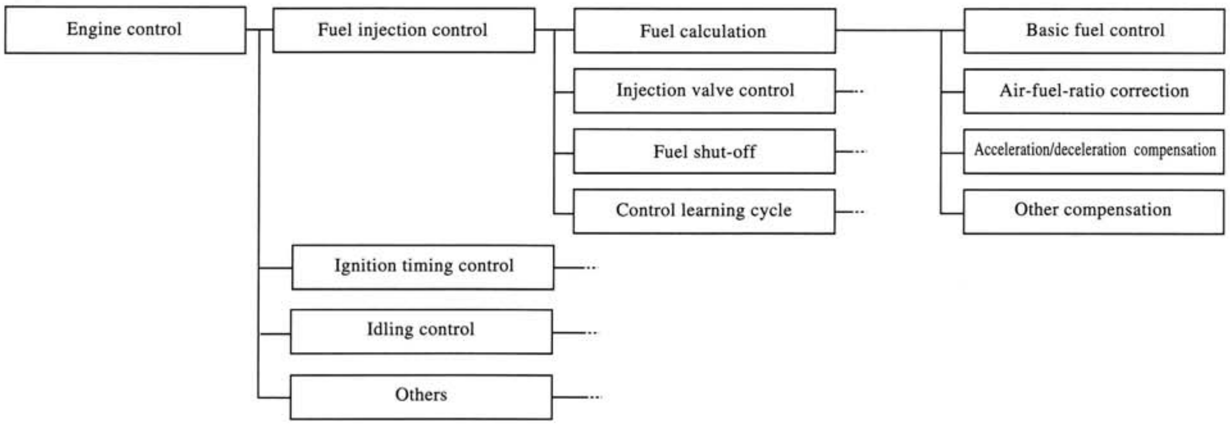
2.1 Effective design

Two important concepts in software production are the "waterfall model" and "structured programming." Program production consists of system design based on analysis, program design, testing, and operation. The waterfall model expresses the overall process of program development.

The structured programming reflects "top-down" program development. The steps in the program production process are divided into function blocks, with substeps done by subdivided function blocks. A divided program block is called a module. A modular program with an independent function is designed independently of other modules. This enables joint program development by more than one designer.

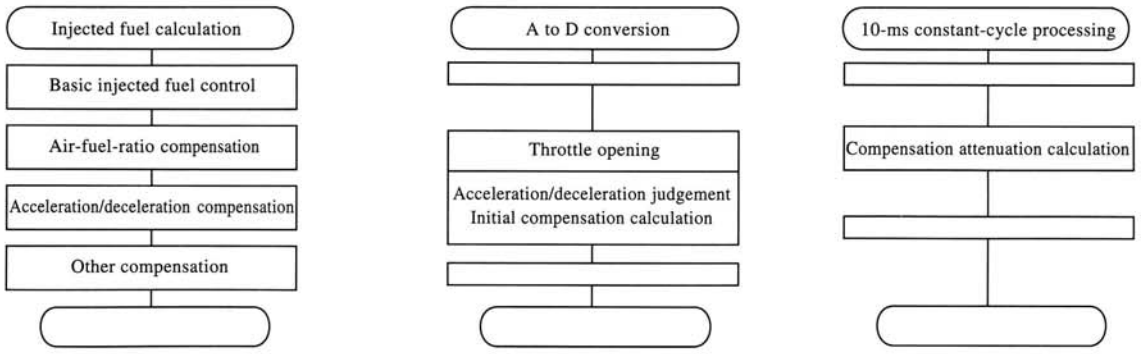
Generally, such development is used with structured languages such as Pascal and C. The basic concepts are, however, applicable to program development in assembler, although some problems must be solved to use procedures in automobile control. Increasingly integrated automobile control requires real-time processing for quick reaction and multiplex control for simultaneously implementing multiple functions. It is difficult to design control programs through structured programming because it involves program configuration arrangement such that processing common to functions is covered by one module and items are divided for well-timed processing by priority.

Such arrangement results in modular programs becoming less independent. The designer must then design programs taking into account relationship to other modules. The degree of the designer's understanding of modules and control specifications significantly affect the outcome. (Figures 1 and 2).



In modular programming, the function division process will basically become a program design process. However, in engine-control, consideration is required in dividing functions into modules and allocating them.

Figure 1. Engine control program


Injected fuel calculation
Basic injected fuel control
Air-fuel-ratio compensation
Acceleration/deceleration compensation
Other compensation
A to D conversion
Throttle opening
Acceleration/deceleration judgement
Initial compensation calculation
10-ms constant-cycle processing
Compensation attenuation calculation

The acceleration/deceleration compensation process is divided into a few routines. Two of these are judgement in the A to D conversion routine and the compensation attenuation calculation in the 10-ms constant-cycle processing.

Figure 2. Program modules of compensation calculation

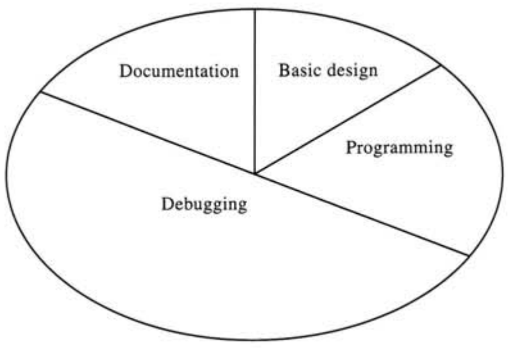


Figure 3. Estimated work allocation of program development

2.2 Debugging problems

The structured or modular programming can make program development more efficient, but these programming procedures do not verify the validity of the developed programs, which must be inspected to make sure they meet specifications. This process is called debugging. If debugging pinpoints an operation not meeting specifications, the program must be corrected from the upstream process on. This is repeated until specifications are met and takes up a large part of program development—more than 50% (Figure 3). In automobile control, a minor error may lead to serious trouble, which requires that programs checking must be especially strict.

Basic debugging consists of checking how each process described in the specification is done. The needed stability and reliability of control, however, require more steps. For example, specifications for the control of an engine's idling speed are applied only to the low revolution region at about 1000 rpm. The actual idling speed may exceed 6000 rpm, however, and debugging must not leave out idling speed control or the closing of the throttle. Even under conditions not specified, the whole region where an engine is operated must be checked for proper engine control.

Checking how each processing in specification is done over the entire range, however, requires very much work not actually covered. Joint design by a group may complicate understanding of overall program operation and cause unexpected problems.

2.3 Program verification (Problem solution)

The above problems originate in the overall complexity of a large-scale program. These problems must be solved in such a way that programs are easier to understand. Basic program design in terms of the waterfall model is characterized by top-down program generation, with frequent feedback between processes. Design is made more efficient by decreasing the amount of feedback or reducing cycle time. This requires a system which can verify a top-down design process. A solution is to indicate programming processing contents and execution conditions relevant to required specifications. This makes it easy to verify operation and locate problems early on.

One of largest problems is related to the difficulty of understanding intermodule relationships. This is partly solved by good program documentation.

A number of program development tools bnased on personal computers and work stations are used with advanced language. Many of these tools, called computer-aided software engineering (CASE), are used in aims design automation and CAD. CAPAS was developed to make program operation easier to understand. It articulates execution conditions and processing contents using arithmetic expressions and familiar notation, eliminating many of the drawbacks associated with top-down program development

3. CAPAS overview

CAPAS makes programs in assembler easy to understand. One instruction in assembler has no meaning for program processing, but a combination does. In Assembler, a flowchart is almost equivalent to a source listing. CAPAS provides a combination of related instruction words, related by arithmetic and theoretical expressions. CAPAS analyzes a program's contents to be stored and branching conditions.

CAPAS also verifies control programs based on the results of program analysis. Assembler itself detects syntax errors (errors in notation and format) during compilation into machine language. CAPAS detects program errors in the contents of execution. Registering a sequence of inspection specification conditions enables errors in module use to be detected. This effectively prevents the recurrence of bugs.

3.1 System configuration

A branching instruction devides a program into two directions of movement. This means that the number of program routines needed for program analysis equals the square of the number of branching instructions. Program analysis must deal with an enormous amount of data, including combinations of unreal conditions and routines not executed depending on the condition. This requires a computer having as large a memory capacity as possible and as fast processing as possible.

In our development work, we use a workstation. It has 32M bytes of main storage and a disk capacity of 390M bytes.

CPU	68030 (25 MHz)
OS	UNIX (System V)
Main storage	32 MB
Hard disk	330 MB

Figure 4. Work station C

3.2 Principles

Assembler compiles mnemonic code into corresponding machine language for the CPU. Mnemonic code expresses basic CPU operation. Except for special instruction systems for control, basic CPU operation is classified into arithmetic operation, substitution, and branching. The arithmetic operation provides data processing for implementing program objects. The substitution provides the entry data to be used in arithmetic operation and the output of the results of the arithmetic operation. Branching is the most important element for program design. In a narrow context, a program aims at data output. The repetition of data input and output and of arithmetic operation leads to the acquisition of data as the final object. Data is stored in memory and the output register. Data is the result of program processing. Conditions imposed on branching instructions until the final data is stored make up execution conditions. Thus, the understanding of contents to be stored and of execution conditions enables the user to understand how a program runs.

Analysis is broken down into ① flow structure analysis, ② stored content of analysis, ③ branching condition analysis, and ④ flowcharts.

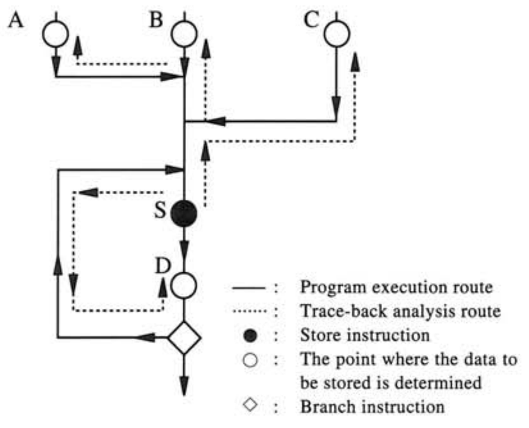
① Flow structure analysis

The analysis of what routes a programmed instruction takes is called flow structure analysis. Branching occurs at absolute and relative addresses and indexes. Simple branching is easily identified from code. Index branching depends on the content of the index register, so the value assigned to the index register must be known. This is determined in the same way as stored contents analysis.

② Stored content analysis

Determining, in the order of instruction execution, how an instruction operates requires that data changes resulting from instruction execution and changes in the CPU state by stored. This requires an enormous memory. It is difficult to know when data is processed, because data is processed where it is most convenient for the program.

CAPAS searches for and analyzes instructions related to the stored data in the direction the reverse of program execution. This search begins at the last stored instruction to be analyzed. This is called trace back analysis. It lets all needed information be gathered efficiently. Actual analysis encounters some problems in impossible routes and changed stacked data and the resulting changed locations to which subroutines return. Improved structure analysis permits the flow of program execution to be traced correctly (Figure 5).


A
B
C
S
D
—— : Program execution route
.... : Trace-back analysis route
● : Store instruction
○ : The point where the data to be stored is determined
◇ : Branch instruction

Starting from S, the analysis proceeds in the reverse direction of program execution. Analysis is done efficiently while collecting only the required data.

Figure 5. Stored data analysis by trace-back method

③ Branching condition analysis

Branching condition analysis also uses the traceback technique. It is important to judge whether an instruction is related to target information and whether all needed instructions have been determined. Each instruction owns data on how its execution affects CPU registers. This data is used for the above judgement. Results of execution are linked by instruction.

④ Flowcharts

Figure 6 shows sample issued results of a CAPAS analysis. Unlike a flowchart generator, the stored data and branching conditions are covered in the flowchart. This makes the program flow and execution contents fairly easy to understand. A flowchart is easier to understand than an Assembler listing. Each program module is assigned a flowchart which describes the conditions of module execution. The integrated, simplified conditions of module execution are related to conditions for branching instructions going though routes to the module (conditions of nonbranching) by logical algebraic operation. This simplification is such that no logical value is determined but algebraic branching conditions are identified. The simplification helps in module programming to determine whether modules are located correctly.

```
          CAPAS
BLOOP     LDD       CNT 10
          CMP 4     # 10
          BCS       LBL1

          CLR       CNT 10
          CLRB      PORT 1, YLED 1
          LDX       #CNT 100

          CALL      SUBCNT

          CMPA      # 10
          BCS       LBL 2

          CLR       CNT100
          SETB      PORT 1, YLED 1

          BRA       LBL 2


LBL 1     CMPA      # 5
          BCS       LBL 3

LBL 2     CLRB      PORT 1, YLED 2

          BRA       LBL 4


LBL 3     SETB      PORT 1, YLED 2

LBL 4     LDX       #CNT10
          CALL      SUBCNT

BLOOPX    JMP       BLOOP


SUBCNT    LDA       O, X

          TBCC      PORT 2, YSW, SUB 10

          INCA      O, X
          STA

SUB 10    RET
```

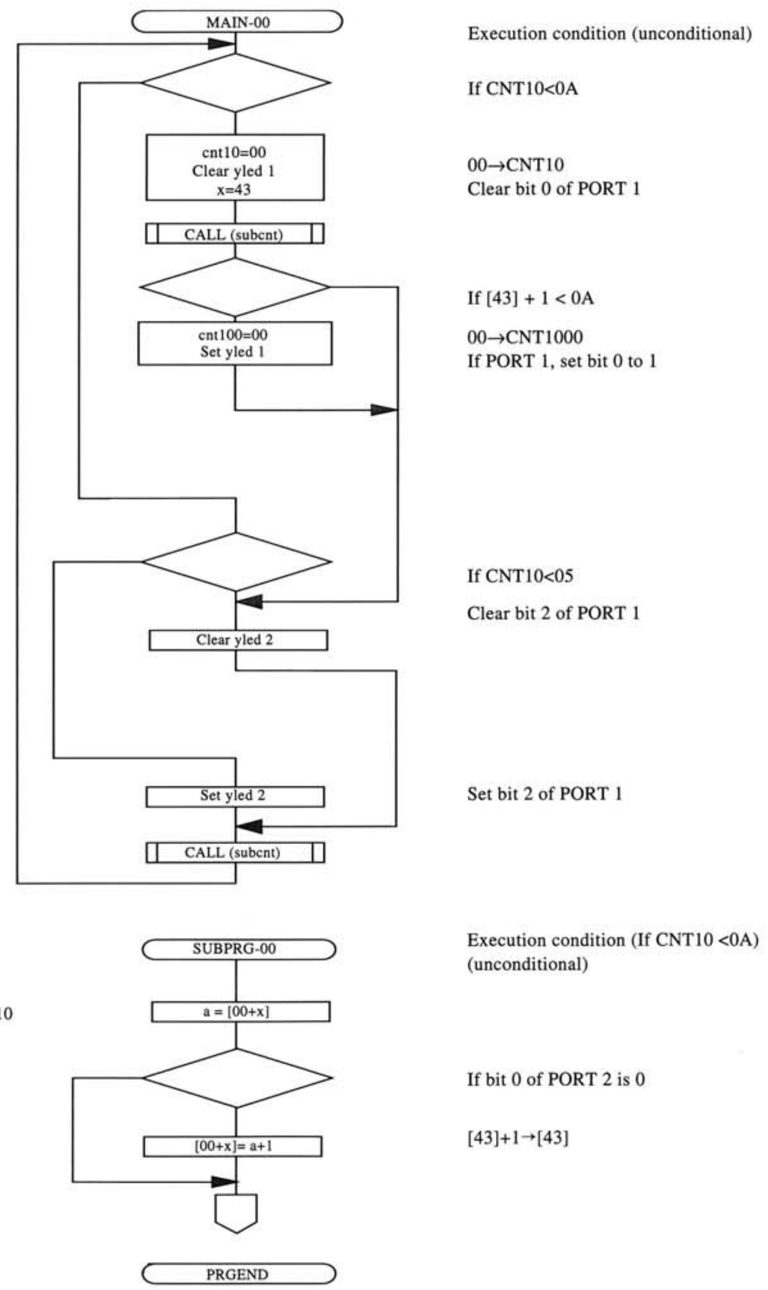


Figure 6. Output sample

4. CAPAS effects

CAPAS has the following features:

① Flowcharts permitting the easy finding of branching contradictory to intended design
② Indication of all branching conditions enabling the user to determine whether a program will run in line with specifications
③ Stored contents for making certain no data contradicts specifications
④ Program inspection preventing inadvertent bugs
⑤ Prevention of modules from becoming black boxes.

Figure 7 shows the transition of debugging time in our engine control program. Debugging time has gone down steadily since the year module programming and CAPAS were implemented.

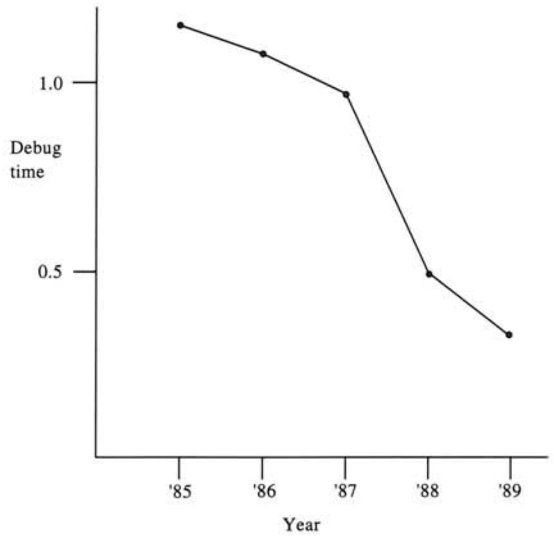


Figure 7. Change of debug time

5. Conclusions

Although advanced languages have begun to be applied in the field of automobile control, their complete adoption requires that problems in processing speed and memory capacity be solved. Under general considerations, it would be advantageous to be able to use languages such as C, but a problem arises in design having to consider machine language and CPU time. Resulting source listings are very difficult to read, although not as hard as those produced with Assembler. Thus, the adoption of an advanced language has its positive and negative aspects. The positive aspect of CAPAS is that it tentatively solves these problems.

Currently, CAPAS can be used for only assembler language, although its basic principles have a much wider application. CAPAS makes the contents of execution easy to understand and enables CPU time to be calculated by conditions, CAPAS will eventually show overall program operation in the form of a tree, with the contents of execution known from CPU output signals at the top and how the related functions are executed lower down. The CAPAS remains to be applied practically because execution conditions are too strict and it runs too mechanical. However, we believe it has great potential in contributing to the design of reliable programs.

Hirotoshi Tono

Entered the company in 1979. He has been engaged in the development of engine-control computers, and currently works in the System Research and Development Department of the Vehicle Electronics Division.

Hisanori Yasugi

Entered the company in 1985. He has been engaged in the development of engine-control computers, and currently works in the Research and Development Department.

Minoru Takahashi

Entered the company in 1970. He has been engaged in the development of electronic equipment for automobiles, and is now in the Research and Development Department.

Hiromi Uchimaru

Entered the company in 1985. She has been engaged in the development of engine-control computers, and currently works in the System Research and Development Department of the Vehicle Electronics Division.