General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

THE JOHNS HOPKINS UNIVERSITY

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

REPORT

NASA GRANT 1442

INTERMITTENT/TRANSIENT FAULTS IN DIGITAL SYSTEMS

BY

GERALD M. MASSON

The Containment Set Approach to Crash-Proof Microprocessor Controller Design*

Robert E. Glaser and Gerald M. Masson
Department of Electrical Engineering and Computer Science
The Johns Hopkins University
Baltimore, Maryland 21218

Abstract

Fault analysis of digital systems has been shown to be possible through the development of a collection of functional level operating states called a containment set. The containment set approach leads to a new method for the implementation of fault tolerance to intermittent/transient (I/T) faults by removing all erroneous elements from the containment set. The achievement of system fault tolerance is then equivalent to proposing a containment set candidate and proving that the candidate is a containment set. The resulting containment set must then be shown to be suitable for practical application by supplying a method to produce each of its elements, and to provide a means to remove all erroneous elements from that set. This paper applies these techniques to microprocessor controllers so as to transform a typical control system into a slightly modified version which is proven to be crash-proof: after the departure of an I/T fault, the inherent design of the system is such that eventual return to the proper control algorithm is assured, assuming no permanent faults occur.

A useful containment set candidate is found for the 8085 CPU with an accompanying proof that it is a containment set. This set is the basis of crash-proof design for the 8085 microprocessor; the same approach can be applied to other microprocessors.

Introduction

Containment set theory was first developed [1] to better understand the effects on digital systems which perturbation by I/T faults can produce. The approach concentrates attention to the upset level, where system function is of prime concern. When the operation of the system can be completely described in terms of a finite set of mutually exclusive functional states, covering all possible transfer functions, this set of functional states is referred to as the *containment set*. All possible system states must cause functional operation of one of the elements of the containment set. This set must include all possible valid functional states of the fault-free system, but this set must also include invalid functional states, not explicitly designed into the system but into which the system can nevertheless be driven by a transient. It was shown in [1] how probabilistic modeling and analysis of a control system's steady state response to transient faults can be made given a containment set, that a suitable form for containment set elements in microprocessor controllers is the program loop, and how erroneous loops (erroneous containment set elements) can result. The modeling and analysis ideas are generalized in [2].

The program loop examples of [1] serve as simple examples indicating how a containment set can be formed. The task of finding all possible loop structures which a microprocessor controller can support is handled in this paper.

* *This research was supported by NASA Grant NSG 1442.*

Upset Theory Applied to Microprocessor Controllers

The digital systems of interest are microprocessor controllers. These controllers typically consist of one printed circuit board, and contain the CPU, several thousand bytes of ROM for program storage, several hundred bytes of RAM for data storage, and I/O devices. Previously developed theory is applied to these systems in preparation for the evolution of crash-proof controller design rules.

The purpose of this section is to produce a finite set as a candidate for a microprocessor controller containment set, and to prove that it is indeed a containment set. To achieve this goal, the ideas of: loop set; erroneous loops; ROM restrict mechanism; special state detector; SAFE ROM; program structures; path diverters; loop structure types; improper subroutines; and STACK WALL are introduced.

Containment Set Characterization

The definition of a useful containment set is the key to the practical use of the developed theory. The containment set elements must define controller functions. The functions of microprocessor based designs are determined by the application program. The control program is as much a part of the controller design as are the hardware details, and any useful approach to fault analysis of such systems should be expected to take into account the particular application program. For these reasons, the containment set approach to the understanding of I/T fault relationships in microprocessor controllers not only includes, but intimately involves, the application program. Therefore, the hardware "system" cannot be analyzed independently from the software; for the hardware is not the system -- the hardware in conjunction with the software is the real "system."

Programs

For microprocessor controllers, programs can be classified as one of two types: those which exit after execution, and those which continuously loop. An *exiting program* performs some calculation, or performs some function, and then terminates. A *loop program*, or simply *loop*, is a program which has the capability of looping indefinitely and the *loop set* {L} is the set of all possible loops defined by a given system. Under some conditions, the program may terminate, but unless it terminates under all possible conditions, it is classified as a program loop. A loop program usually monitors inputs or processes input data and transfers results to the output. Often, a loop monitors some input condition, waiting for the arrival of some specified state before exit. In the following, only loop programs will be considered. This is not particularly restrictive for microprocessor controllers, since process control software [3] is such that most often controller programs are indeed loops. Also, a number of exiting programs executed in sequence can result in a single loop program.

Within this framework of a microprocessor controller, the loop programs are natural choices for functional level program states. A program loop defines I/O relationships. The processing between the reading of input sensors and the writing to system actuators sets the system transfer function. If a loop contains no outputs, the I/O relationship reduces to the trivial. The operational status of the controller can be given as a specification of the particular loop program currently under execution. Moreover, in addition to valid loop programs that are explicitly written for the application, there can be erroneous embedded loops to which the system can be driven by a transient fault. The loop set is then the collection of all valid and erroneous loop programs. Before returning to these erroneous loops, the interaction between transient faults and the loop set will be discussed.

Given that the controller is executing some loop program, $L_i \in \{L\}$, upsets can be characterized in three ways:

Class 1 Upset: Data Change. A transient fault is said to produce a data change upset when that fault changes data values being transferred and/or stored, but loop program L_i continues to be executed.

Class 2 Upset: Program Bump. A transient fault is said to produce a program bump upset when that fault causes a temporary divergence from loop program L_i, but the controller eventually returns to that program.

Class 3 Upset: Program Transition. A transient fault is said to produce a program transition upset when that fault causes the controller to jump from the execution of L_i to the execution of $L_j \in \{L\}$, $i \neq j$.

The data change upsets and program bump upsets are the least significant of the three upset classes because their effects on the controller's operation are temporary, and the controller itself either continues, or returns within a finite time to the execution of the proper loop program. In many control applications, such as those in which a mechanical device is being operated, temporary data changes or program bumps usually occur at a rate exceeding the device's capacity to respond. On the other hand, a program transition upset resulting from a transient fault is a steady state operational deviation which can have most serious consequences. Indeed, the result of a transition into an erroneous embedded loop is usually referred to as a system crash. These comments, of course, are a reiteration of the justification for the importance of operational level containment sets put in terms of microprocessor controllers.

	ADDRESS	CONTENTS
NORMAL EXECUTION	N	OP-CODE
ERRONEOUS EXECUTION	N+1	DATA
NORMAL EXECUTION	N	OP-CODE
ERRONEOUS EXECUTION	N+1	DATA 0
ERRONEOUS EXECUTION	N+2	DATA 1
ERRONEOUS EXECUTION	N	DATA TABLE
ERRONEOUS EXECUTION	N+1	DATA TABLE
.	.	.
.	.	.
.	.	.

FIGURE 1. ERRONEOUS LOOP INSTRUCTION EXECUTION

The loop set is proposed as a candidate for a containment set. Since the containment set is defined in terms of loop programs, a complete investigation into the ways in which these loops can be formed is necessary. It will be shown that there are a variety of program structures which can support program loops. These structures may be either valid or erroneous loops.

Erroneous Loops

In addition to valid loop programs, the existence of erroneous embedded loop programs has been mentioned. A transient fault can cause program execution to begin at any memory location. This can cause incorrect execution of a valid loop program, if more than one valid loop program exists, or it can cause the execution of an *erroneous loop* program which is embedded in the application program (program crash). It therefore is appropriate that a simple example of the existence of such phenomena be given. Erroneous program loops exist because of multiple byte instructions and data storage, as shown in Fig. 1. If execution erroneously begins at a data location, the data will be interpreted as an op-code, and execution of an erroneous loop program can begin.

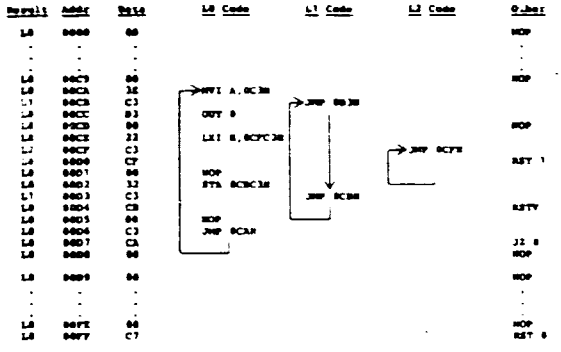


Figure 2. An Erroneous Loop Example for the 8085

An example program for the Intel 8085 8-bit CPU [4] is given in Fig. 2. The loop set {L} consists of one intentional loop, L0, and two erroneous loops, L1 and L2. The entire contents of the 256-byte ROM are shown, from address 0000 to 00FF. Execution of L0 normally begins at address 00CA, where the 3E is the op-code for the MVI instruction. L0 consists of seven instructions, and if execution begins at the first byte of any of these instructions, execution will remain in L0. These addresses are: 00CA, 00CC, 00CE, 00D1, 00D2, 00D5, and 00D6. If execution begins at 00CB, the data value C3 will be interpreted as a JMP instruction. The result, therefore, is the execution of the erroneous loop, L1. Execution at 00D3 also results in the execution of loop L1. Execution at 00CF produces a one instruction loop, L2. If execution begins at addresses 0000 through 00C9, each 00 will be interpreted as a NOP instruction, leading to loop L0. Similarly, addresses 00CD, and 00D8 through 00FE result in NOP's. Address 00D0 is interpreted as RST 1, which sends control to the NOP block at the head of the program. At address 00D4 is found the data CB. This is not an Intel defined 8085 instruction. It has been found, however, that upon execution of this op-code, an instruction called RSTV, restart on overflow, results [5]. Execution either continues or control passes to the top block of NOP's, in either case the result is loop L0. At address 00D7, the CA is interpreted as JZ, so control either goes to address 0000, or to the trailing NOP's. The last memory location contains C7, interpreted as RST 0, which sends control to address 0000. The RESULT column of Fig. 2 shows which loop is reached if execution begins at each memory location. In this example, there are only three memory locations from which execution will result in an erroneous loop program.

These erroneous jump loops are but one type of loop program. As will be seen later in this section, other program structures permit other forms of both valid and erroneous loops.

Hardware Requirements

The set of loop programs has been proposed as a containment set for microprocessor controllers. Up to this point, it is not clear that this loop set satisfies the containment set requirements: namely, that they form a complete, finite, mutually exclusive set of all possible operational states. The set of all possible valid and erroneous loop programs is certainly mutually exclusive. However, it must be further shown that this set is finite and that it is possible to determine each element of the set. It is also necessary to show that the loop set describes all possible operational states of the controller. These requirements will not in general be met by an unmodified microprocessor controller. However, it will be seen that it is possible to satisfy these requirements through the modification of typical microprocessor controllers.

The realization of a controller in which upsets can be characterized as loop program transitions requires that some specific, but not excessive or unreasonable, attributes be designed into the system. First of all, if programs are permitted to execute from RAM, and since the contents of RAM can be changed at any time both during program operation and by a transient fault, there would be a nearly infinite set of possible loop programs which could be executed from RAM. This is because in this case the RAM contents are the instructions themselves; it is possible for any sequence of instructions to be placed into RAM and executed, and the set of loop programs consists of all possible loop programs which can be written within the CPU instruction set. Therefore, it is necessary that the application program be stored in the controller's ROM, and the program execution be forcibly restricted by hardware to that ROM. This *ROM restrict mechanism* is not a program restriction, since this already is the case for microprocessor controller designs. Since all of the valid loop programs are defined by the application program, the ROM restriction guarantees that the set of valid loop programs – a subset of the containment set – remains fixed during system operation, as long as no permanent faults occur.

The set of erroneous loops must be finite, fixed, and determinable. Due to the ROM restriction, only program execution from the ROM need be considered. However, since small microprocessor systems often use incomplete address decoding, in such cases the ROM can be activated from many addresses other than the ones which are intended, creating many images of the ROM. In normal operation, this does not matter, since the control program never sends control out of the primary ROM address space. With the addition of faults, it is possible for these images of the ROM to be accessed, and the ROM restriction will not prevent this occurrence. This can be permitted, but the complete set of addresses which can access the ROM must be known. In addition, the complete ROM contents must be known – there is usually unused storage space which is not defined, and not necessary for fault-free program operation. With a complete ROM specification, the set of erroneous loop programs is finite, fixed, and determinable. Therefore, since the containment set is the union of the valid and erroneous loop sets, and each is finite, fixed, and determinable, the entire loop set is finite, fixed, and determinable. All that remains to be shown to conclude that the loop set is a containment set is that it covers all possible operational states.

The loop set describes all possible operational states while the processor is executing instructions. However, there are states the CPU can enter which can halt the execution of instructions. These states will vary from processor to processor, but for example, will usually include the halted state. Hence, any special function that is included in a processor's operational capabilities, and which can prevent normal program execution, must be removed. If this is done, then the loop set becomes complete (the CPU cannot remain in any state not in the loop set), and qualifies as a containment set.

The additional hardware, then, must include the ROM execution restriction and a *special state detector*. The special state detector flags entry into other prohibited states. If a bad state is entered, something must be done to escape it. A nonmaskable trap input can be used to force exit from these special states, or a simple system reset may likewise be used. Regardless of the implementation details, it should be appreciated that, together, the two additional hardware sections have a complexity on the order of two to three integrated circuits.

It should be noted that with the addition of hardware to the controller, the system is inherently changed. It is possible for the new hardware to create new states which can prevent program operation. The hardware additions must be analyzed and modified if necessary to verify that this is not the case.

With these restrictions, the processor is guaranteed to be executing instructions; those instructions are guaranteed to be contained in ROM; the ROM address space is completely defined; and therefore, in the steady state, the processor must be executing one of the programs in the loop set, so the loop set is a containment set.

An alternative to determining all possible erroneous loops is to prevent their execution with hardware. This can be done with the addition of what will be called a *SAFE ROM*. The SAFE ROM is a 1-bit wide memory, with the same number of locations as the main ROM (where the control program is stored). The single additional bit is used to distinguish between valid and erroneous instruction fetches. The first byte of multiple byte instructions is marked as valid; all other locations – data, addresses, and lookup tables – are marked as erroneous. The ROM restrict hardware must then verify through the SAFE ROM that a valid instruction is being fetched. If not, a trap interrupt or a reset must be issued. SAFE ROM hardware supplies the option of trading off hardware for the additional complex program analysis needed to determine the erroneous loop set. The SAFE ROM performs the function of limiting the *capabilities* of the operating software, used here at a lower level than as first defined in [6].

Program Structures

The problem remains of finding all possible valid and erroneous loops, given complete ROM contents and ROM address specification. This problem will be attacked by finding all *program structures* which can possibly result in a loop. Microprocessors execute instructions in sequence, until an event intervenes to divert the path of the program. The search for all loop program structures will center on the concept of a path diverter. A *path diverter* is defined as any event or condition which can divert program flow from the normal incrementing address sequence. Path diverters may either be hardware or software based.

Hardware Path Diverters

For accuracy and completeness, the following will be restricted to the 8085 CPU. Similar, if not identical, analysis can be done for other microprocessors.

For the 8085, the following hardware features can divert program flow: clock stop, halt, ready, hold, and interrupts. The clock oscillator must be such that it unconditionally oscillates. This is an often overlooked but very important issue in any microprocessor design, but it is particularly significant for fault tolerance. Fortunately, this source of path diversion can be avoided simply by stringently adhering to the manufacturer's specifications. Peripheral circuitry which utilizes the hold or ready inputs must be configured such that a reset pulse assures normal operation resumes should some unplanned lockout take effect. (Such as a poor bus arbitra-

tion scheme which would grant the bus to neither device if two devices contend for the bus at the same time.) This is normally automatic. If it is assumed that the hardware additions from the previous section are incorporated (special state and bad fetch detectors, and possibly a SAFE ROM), then program fetches are guaranteed to occur within a specified period of time, or a reset occurs. Therefore, halt, hold, and ready features cannot stop the program execution in the steady state.

This leaves interrupts to be considered. Normally, when an interrupt occurs, an interrupt service routine is executed, and program flow returns to the interrupted program. When this happens, the program path has been diverted, but only in one sense, for the program flow eventually returns to the place at which the interrupt occurred, continuing the original path. A normal interrupt sequence need not be considered a path diverter for the purposes of loop analysis, because any program loop which contains a normal interrupt sequence will still be a program loop without the interrupt sequence. Therefore, the interrupt sequence must be "normal" in the sense that the interrupt service routine must return. The service routine is a subroutine, which also can be called from within the program, so with the understanding that it will be considered a subroutine, and that subroutines will be handled in the section on software path diverters, this hardware path diverter can be bypassed – with one exception. A further requirement of an interrupt service subroutine is that it executes in less time than the time between successive interrupts. In a properly operating program, this will always be the case. However, in some special cases, additional provisions must be made to enforce this. If the device which is the source of the interrupts is a programmable peripheral device which has modes of operation which can allow the interrupt interval to decrease under program control (and therefore possibly as a result of a transient fault) steps must be taken to ensure that the interrupt interval is restored after each interrupt. An example of this would be a programmable divider driving an interrupt input. Should the programmable divide ratio be changed such that the period between interrupts becomes less than the service routine execution time, then as soon as the interrupt routine exits, another interrupt will immediately occur. This causes a program loop. In such a case, if within the interrupt service routine the divide ratio of the programmable divider is reinitialized to the proper value, then this interrupt loop can never occur in the steady state.

For the case of external sources of interrupts, the hardware should be examined to verify that no modes of operation exist which can cause the period between interrupts to decrease below that of the interrupt service routine execution time.

Software Path Diverters

Examination of the instruction set of the microprocessor of interest gives the instructions which can divert normal program flow. A problem exists with undefined op-codes. Two options may be employed with these op-codes. A SAFE ROM may be utilized, assuring that only valid instruction fetches are made, which would never include undefined op-codes. Alternatively, the undefined op-codes can be defined by the user. For the 8085, all op-codes not defined by the manufacturer have since been defined by others. Investigation has shown that these "undefined" op-codes may reliably be employed [5]. Therefore, the SAFE ROM approach need not be used with that CPU.

In the 8085, the software path diverter instructions are in the following categories: jump, call, and return. The jump category includes the unconditional JMP, and the ten conditional jumps (including the "undefined" jumps). The special instruction PCHL (jump to the address specified by the HL register) is also in the jump category. The call category includes the unconditional CALL, and the eight conditional calls. The nine RST (restart, usually used for interrupts, but sometimes used in place of CALL) instructions (including "undefined" RSTV – restart on overflow) are single byte calls, and are also in that category. The return category consists of the unconditional RET (return from subroutine), and the eight conditional returns. A program structural analysis centering on these path diverters follows.

Program Structural Analysis

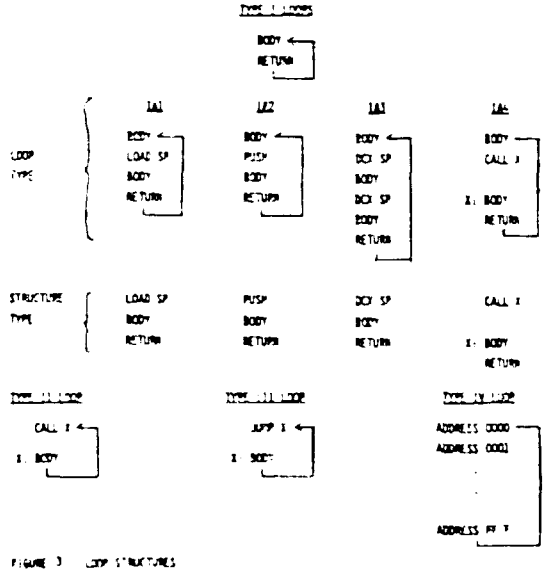
The assumption that the processor continously executes instructions is satisfied through hardware requirements. Therefore, in the steady state, a program loop must exist. More precisely, this implies that there exists at least one address containing an instruction such that program flow eventually returns to this address after execution of this instruction. It is desired to determine all possible program structures which can permit such a loop to exist. These will be found by considering the four *loop structure types* containing the various path diverters.

Type I – Loops Containing Returns

This loop type contains a path diverter of the return category. As shown in Fig. 3, there is a body of code leading to a return instruction, which then sends program flow back to the beginning of the preceding body of code. The return instructions divert program flow by changing the program counter to an address found at the top of the stack; therefore, the stack pointer plays an important role in the path selection. Type I loops will be divided into (A) those which leave the stack pointer unchanged between loop iterations, and (B) those which cause the stack pointer to be changed between loop iterations.

Considering first the loop type IA, it should be noted that the execution of return instructions increases the stack pointer by two. In order for there to be no net change in the stack pointer in one loop iteration, it is therefore necessary for either the stack pointer to be decreased by two in the body of code preceding the return instruction, or for the stack pointer to be initialized at least once per loop. Those loops which load the stack pointer are classified as type IA1. There are only two instructions which can load the stack pointer: LXI SP, and SPHL. It is evident that for loop type IA1 to exist, structure type IA1, shown beneath loop type IA1 in Fig. 3, must exist.

Considering next loop types IA which reduce the stack pointer by two prior to the return instruction, there are three ways this can occur. All PUSH instructions decrement the stack pointer twice, so loop type IA2 results. Structure type IA2 must exist for loop type



IA2 to exist. The stack pointer may directly be decremented twice, resulting in loop type IA3. Structure type IA3 is necessary for the existence of this loop type.

For each instruction in the call category, the stack pointer is decremented by two. Loop type IA4 results, for which structure type IA4 must exist. It should be obvious that this structure is that of a standard subroutine.

Loop type IA4 can be further divided into those in which (a) return is as expected, and those in which (b) return is incorrect. For the first type (IA4a), the call to subroutine X is not necessary to form the loop. Even if the "CALL X" instruction is bypassed, the loop still exists, since upon a normal subroutine return execution continues at the instruction after the "CALL X" instruction. Hence, this return instruction is not diverting program flow, it is not the source of the loop. The loop structure remaining when the "CALL X" is removed must itself be one of the loop types categorized which does not contain the specific return instruction being considered. For this reason, in the search for all loop structures, type IA4a need not be considered.

Loop type IA4b is such that the return is not to the correct location. This can happen for two reasons: (1) the stack pointer does not point to RAM, or (2) the stack pointer does point to RAM, but the contents of that RAM addressed by the stack pointer – the stack space – is modified between the original "CALL X" and the return statement. Subroutines which modify either the stack space or the stack pointer in such a way that return is not to the instruction after the call will be classified as *improper subroutines*.

To simplify the search for loop types IA, the role of call category instructions encountered in the segments of code listed as "BODY" must be considered. If the CALL leads to a code segment which contains a return category instruction, this falls under loop type IA4. If the CALL leads to a code segment which does not contain a return, then this subroutine is also classified as an improper subroutine. Hence, for the consideration of "BODY," call instructions may be ignored, since their existence can only lead to a structure already covered, or to an improper subroutine.

In summary, then, for a loop of form IA to exist, one of the structures IA1, IA2, IA3, IA4b1, or an improper subroutine must also exist.

Loop programs of form IB contain a return category instruction, and modify the stack pointer each iteration. These loops will cause the stack pointer to traverse the entire address space. As an aid to the discussion of type IB loops, the STACK WALL is defined.

Let l_d be the largest positive value within a body of code not containing a structural loop by which the stack pointer can be decremented, and l_i the largest by which the stack pointer can be incremented. Let $W=1+\max(l_d-2,l_i+2)$. The *STACK WALL* consists of a contiguous block of memory locations of length W, such that for any consecutive pair of bytes within that block, an address is formed which is guaranteed not to be within a type IB loop.

Each iteration of a type IB loop changes the stack pointer by $SP=SP+K$, where $-(l_d-2)\leqslant K\leqslant(l_i+2)$. If a STACK WALL exists, then any type IB loop must either contain a structural loop (permitting the stack pointer to be changed out of the limits set by K) or eventually cause the stack pointer to address locations entirely contained within the STACK WALL. When this occurs, the return address will be fetched from within the STACK WALL, and program flow will exit the potential type IB loop, effectively breaking the loop.

In summary, given the existence of a STACK WALL, type IB loops cannot exist in the steady state, or are covered by another type of loop structure.

Type II – Loops Containing Calls

Type II loops contain call category path diverters. If the subroutine called leads to a return type instruction, it will be covered as a type I loop. Therefore, only calls to subroutines not leading to returns need be considered. If the call does not lead to a return, it must lead to a loop: if that loop does not contain the original call instruction, it is not a type II loop. If the subroutine leads to the call instruction, it will be classified as an improper subroutine. Therefore, the existence of type II loops is dependent upon the existence of an improper subroutine.

Type III – Loops Containing Jumps

Type III loops contain jump category path diverters. Such potential loops containing return instructions need not be considered, as they are covered as type I loops. Calls can be bypassed in a search for type III loops, since if they act as path diverters, they will be covered as type I ~ II loops. Therefore, type III loops consist of simple branching structures, and the loop itself is considered a program structure.

Type IV – Loops Without Path Diverters

Type IV loops contain no path diverters. Actually, there is only one possible type IV loop, and it loops by accessing each memory location in the address space. This can only occur if the entire ROM contains no path diverters at all (the program counter eventually overflows back to the start of the ROM); this never occurs in actual practice.

In summary, with the existence of a STACK WALL, and the lack of improper subroutines, any program loop must be of types IA1, IA2, IA3, IA4b1, or III. Figure 3 depicts all of these loop types. An improper subroutine modifies its return address or does not lead to a return statement. This can occur if it contains an internal loop, or calls another subroutine which calls the improper subroutine.

This program structural analysis will be the basis for the design of fault tolerant controllers developed in the next section.

Fault Tolerant Controller Implementation

Upset theory applied to the 8085 CPU will now be used to produce a new design approach for transient fault tolerant controllers. The detailed program structural analysis plays an integral part in the implementation method. Design rules – both hardware and software – will be supplied, such that a controller so constructed can be shown to have no steady state loop structures which are not designed-in, and those loop structures which are designed-in can recover from transient faults. This implementation produces a controller which is *crash-proof*, where *crash* is considered to be any situation which causes the processor to permanently cease execution of its intended function.

In order to provide a base from which to evolve this design approach, and to test its practicality, an experimental system was developed. This system also served as a testbed with which to assess the difficulty or ease of the application of this approach, and provided an indication of the number and types of erroneous loops which could be found in real systems.

Design Rules

The design rules for transient fault tolerance cover both hardware and software aspects of the controller. It will be seen that for a large number of applications these additional requirements are not overly restrictive.

Hardware

The CPU must unconditionally execute instructions. This is achieved with the addition of special state and bad fetch detectors. Program execution must be restricted so that the instructions reside in ROM. As an option, program execution can be checked with the addition of a SAFE ROM.

Circuits which satisfy the requirements imposed by these hardware design rules have been developed for the 8085, and only two inexpensive integrated circuit packages need be added to a standard design not considering faults. If a SAFE ROM is further desired, then that ROM increases the additional chip count to three.

Software

The application program must be written such that program structures IA1, IA2, and IA3 are not used. This is because the stack pointer is of necessity in RAM, and these structures can lead to loops dependent upon the contents of the RAM area used as stack space; these loops cannot otherwise be prevented. This requirement is not particularly restrictive, with the possible exception of IA2. Sometimes, when a program variable jump is required, the following instruction sequence is used: PUSH H, RET. This is structure IA2. This structure can be permitted if the HL register is checked after the PUSH and before the RET to verify that it contains a valid address, and that jumps to that address cannot lead to an unknown loop. In most cases, this structure can be replaced by status flags.

A special case of type III loops is that of the PCHL instruction. This instruction must not be used, for it is possible for the HL register to contain the address of the PCHL instruction; if this occurs, then a loop results involving only one instruction, and there is no possible way to verify that this cannot occur. However, the PCHL instruction performs precisely the same function as the PUSH H, RET sequence, and may be replaced with the sequence described above.

A STACK WALL must be included. A simple way to handle this is to leave all unused memory locations in the program store ROM either 00 or FF. In the first case, any return address retrieved from the STACK WALL will cause execution to begin at address 0000. This is the reset location, and by definition, must break any type IB loop. If the STACK WALL contains FF's, execution will begin at address FFFF, and if this address does not access the ROM, the ROM restrict hardware will force a reset, also breaking any type IB loop.

Improper subroutines must not be permitted. The return address must remain intact. Subroutines must not directly modify the stack pointer or stack space. In order to guarantee the latter, wherever a memory store instruction uses a variable address in a register for the store location, that register must be checked to verify that it is not within the stack space. A subroutine must not call itself. Interrupt service routines are treated as subroutines.

To eliminate IA4b1 loops, returns from subroutines must verify that the stack pointer is pointing to stack space in RAM. Instead of performing a return instruction, a jump must be made to a special return routine. This routine checks the stack pointer, and if it is a valid address, a return is executed. If an invalid address is found, the stack pointer must be reloaded, and a program restart made. This restriction is perhaps the most limiting, since it increases the execution time significantly for return statements. Still, there are a large number of applications where the CPU is not running near its limit of speed, and for these applications, this requirement would not cause a problem. Eliminating all returns except the one in the RET routine removes any possibility for loop type IA4b1 to occur.

All possible type III loop structures should be examined. Loops which are intended to be temporary must be verified to ensure that under all circumstances, they are temporary. This can always be accomplished with the addition of a loop counter, if necessary. An example of a subroutine which originally is not fault tolerant, and its modified version, is shown in Fig. 4. The unmodified version can form an endless loop if input variables are invalid. Determination of the maximum execution time before exit for each loop can provide an upper bound for the overall upset recovery time.

A) Routine Correct for Define' Inputs, But an Undefined Input Causes Endless Loop

```
;BIT POSITION TO BINARY CONVERTER ROUTINE
;
;ENTER WITH A SINGLE BIT SET IN REGISTER A
;EXIT WITH REGISTER B CONTAINING THE BINARY
;POSITION OF THE SINGLE SET BIT.
;DESTROYS REGISTER A.
;
        MVI     B,-1    ;INITIALIZE REGISTER B
LOOP:   INR     B       ;BUMP COUNTER
        RRC             ;ROTATE REGISTER A RIGHT
        JNC     LOOP    ;CHECK FOR CARRY SET
        RET             ;RETURN FROM SUBROUTINE
```

B) Corrected Subroutine

```
;BIT POSITION TO BINARY CONVERTER ROUTINE
;
;ENTER WITH A SINGLE BIT SET IN REGISTER A
;EXIT WITH REGISTER B CONTAINING THE BINARY
;POSITION OF THE SINGLE SET BIT.
;DESTROYS REGISTER A.
;
;NOW WITH GUARANTEED EXIT.
;
        MVI     B,-1    ;INITIALIZE REGISTER B
LOOP:   INR     B       ;BUMP COUNTER
        ORA     A       ;SET IF REGISTER A=0
        RZ              ;YES, ABORT AND RETURN
        RRC             ;ROTATE REGISTER A RIGHT
        JNC     LOOP    ;CHECK FOR CARRY SET
        RET             ;RETURN FROM SUBROUTINE
```

Figure 4. Effects of Out of Range Input Variables

All intentional steady state loops must guarantee all operating assumptions. This can be accomplished by calling a check routine from each of these loops. The check routine should guarantee that all I/O devices are programmed properly, interrupts are enabled and unmasked (if used), and any assumptions that the program makes on variables for proper operation are fulfilled.

Exit must be guaranteed from all possible interrupts. Unused vectored interrupts should jump to the start of the program. Interrupts which are generated from a programmable divider must assure the integrity of the divide ratio within the interrupt service routine.

The preceding software requirements must be met, considering both valid and erroneous loops if a SAFE ROM is not used. The appearance of any banned structure as a portion of an erroneous loop usually results from specific address values being interpreted as op-codes. Erroneous calls to erroneous subroutines must be investigated to verify that that erroneous subroutine is not an improper subroutine; if it is, it must either be changed, or the erroneous call removed. To remove erroneous structures, the program should be shifted one byte at a time, until a shifted version results in no erroneous loop structures. There is no guarantee that this is possible; however, programs of length 1K to 2K should pose no real difficulty. For large programs, a SAFE ROM is probably desirable.

Table I summarizes these transient fault tolerant software design rules. These software requirements may appear difficult to employ, but with appropriate software aids, the problem is greatly alleviated.

TABLE I

Crash-Proof Software Design Rules Summary

1. Program execution from RAM is prohibited.
2. Each flows temporary loops must be guaranteed.
3. Stable loops must include a call to a check subroutine, which guarantees all assumptions the program requires for continued execution. (These include assumptions on I/O, interrupts, and variables.)
4. Returns from all possible interrupts must be guaranteed.
5. Subroutines cannot call themselves.
6. Subroutines cannot modify the stack pointer or the return address.
7. Within subroutines, memory store instructions which permit a variable store address must guarantee that the register used as the address pointer cannot point to the stack space.
8. Instead of using RET or RCN instructions, a JMP or JCN to a special return routine which guarantees that the stack pointer points within the stack space before returning must be used. (The return routine contains the only RET instruction in the entire program.)
9. A STACK WALL must be added. (Leave unused ROM space as 00 or FF.)
10. The PCHL instruction cannot be used.
11. Either a SAFE ROM must be used, or all erroneous loops and erroneous calls to addresses which, when considered to be subroutines, do not satisfy either Rule 5, 6, 7, or 8 must be removed.

Software Tools

Two programs were written to ease the implementation of the software requirements for fault tolerance: SAFE, and LOOP. The SAFE program accepts as input assembly language source code (a text file). It determines which memory locations contain the first byte for each instruction. This output can be used directly as the SAFE ROM contents. It is also used as input to the program LOOP.

LOOP is a tool used to locate program structures (developed in section three) which can lead to loops. It accepts as input object code for the application program of interest, and optionally a safe file generated by SAFE. The latter is used to permit classification between valid and erroneous instructions.

While conducting a search for a specified request, LOOP spans all possible trees created by conditional branches. In the loop search mode, it lists all type IA1, IA2, IA3, and III structures. Some of the structures are valid, and some are erroneous. The valid type III structures must then be verified to ensure that those loops not intended as steady state loops have an unconditional escape mechanism. There should not be *any* valid type I structures. The erroneous structures of all types must be removed.

LOOP can be commanded to list all subroutine calls. This is used to search for erroneous calls. Each erroneous call found must be checked: if it leads to an address not contained in ROM, nothing need be done (the hardware will handle it); otherwise, that erroneous subroutine must be checked to determine if it is an improper subroutine, in which case, either the erroneous call or the erroneous subroutine must be changed. As an aid for the verification of erroneous subroutines, a disassembler option is included in LOOP.

A search can be made for all memory store instructions which utilize variable store locations. This is used to verify that the programmer has caught all such instructions in subroutines, and has taken steps to eliminate type IA4b2 structures.

With the interactive use of the loop search program, the programmer can effectively and efficiently produce a modified application program to achieve fault tolerance.

Fault Tolerant Controller Test System

The test system is self-contained and consists of two microprocessor controllers, power supplies, and a fault source. It was used to verify the viability of the design rules developed for fault tolerant controller implementation. Measurements of program execution were taken with a logic analyzer and an oscilloscope.

A noisy power supply was chosen as a reasonably realistic fault source which can be controlled in the laboratory environment. The testbed controllers are 8085 based and utilize a total of six standard integrated circuits, exclusive of the additional hardware (two or three integrated circuits). The controller function is to read the input switches and write appropriate displays to the output lamps.

Crash-proof versions of the original control program were written and exercised in the test system. No permanent program crashes were observed.

Conclusions

Design rules have been developed for microprocessor controllers to provide tolerance to transient faults at a functional level. It has been proven that adherence to these rules will produce a crash-proof controller: all possible program loops are known, each corresponds to a valid mode of operation, and continued processor execution is assured. To accomplish this end, a very small number of additional components are required and the increase in software complexity is low. The testbed controllers required from 8 to 14 percent increase in program size to satisfy the software design rules. This approach to fault tolerance without the use of massive redundancy must be viewed in the proper perspective. It is not intended as a replacement for high reliability data intensive applications, which would normally require duplication or triplication of hardware. It is, however, applicable to the much larger class of systems which do not have such severe requirements, but which can greatly benefit from a guarantee that the system will continue to operate in the long term. Bounds on recovery time can be found through loop analysis, and this can be lowered if desired with the addition of a watchdog timer activated by the check routine.

Testbed Results

Complete adherence with the design rules produces a totally crash-proof controller. The questions are sometimes raised as to how prevalent various loop structures are in the software, and how often a controller will be driven to these loops. With regard to the first question, after loop analysis of approximately a dozen program versions, the presence of erroneous loop structures was observed in many of them. Most of the observed erroneous structures were of the stack varieties. One version contained a PCHL instruction, and another a jump loop. The answer to the first question is that these erroneous structures are quite common, though it is possible to find program versions which are free of such structures.

To respond to the second question, it first must be noted that simple intuition concerning the likelihood of various loops is not at all reliable. Transitions into erroneous loop programs which are intuitively highly unlikely were observed.

The test controllers were run with various combinations of hardware and software. It was found that the removal of either the halt detect or ROM restrict mechanisms caused some mode of program crash. Unmodified program versions were observed to crash with the hardware restrictions in place. Therefore, it has been found that practical systems require both the hardware and the software modifications that have been shown theoretically to be necessary.

Comparison with the Watchdog Timer

Probably the simplest and most prevalent external hardware addition used for fault detection is a *watchdog timer*. In normal operation, the microprocessor periodically pulses the external timer. The software is written such that the processor is guaranteed to pulse the timer before a specified time elapses. The timer is retriggerable, so that under normal operation it never times out. Should the processor fail to trigger the watchdog within the allotted time period, the timer initiates recovery action, which can be as simple as resetting the processor, or as complex as calling a test program which thouroughly exercises the entire system and logs the results. Watchdog timers are discussed in [7], an implementation can be found in [8], and discussion of the effectiveness of watchdog timers can be found in [9,10].

The containment approach must be compared with the watchdog timer technique, which is also used to prevent crashes in the long term. The watchdog timer has been found to be effective in many applications -- however, it suffers from a major drawback: typically, it is not provably complete. When used, there is no assurance that there does not exist some loop program which can trigger the watchdog, yet not maintain correct system functioning. The techniques developed in this research can be used to validate such watchdog timer approaches to tolerance. However, without the ROM restriction, no such validation can occur, since there is always the possibility that a loop can be set up in RAM, activating the timer, and not performing the system function. With a loop analysis, and a ROM restriction mechanism, then, the watchdog timer technique can be validated. Each possible loop structure need only be inspected to verify that erroneous loops do not trigger the timer. This approach uses the same number of parts as does the containment strategy: the ROM restrict mechanism, and a retriggerable timer. The only advantage of using these new tools to validate a watchdog timer approach would be to simplify the software changes. These software changes have been shown to be relatively easy to achieve with small controllers. A combined loop analysis/watchdog timer approach could be useful for larger systems. In a sense, the watchdog approach can be seen to be a subset of the containment approach.

Watchdog timers typically operate with a long time interval. As a result, they reset the system relatively long after the system has crashed. The strict containment approach reacts much faster than the watchdog approach, since the only timer (fetch detect/halt restrict) is on the order of a single instruction time. This shorter timer period, and the immediate response to an erroneous fetch cause these hardware forced resets to respond much quicker to a fault than would a watchdog timer and can have the effect of restoring system operation before errors can propagate as far as with a watchdog approach.

In summary, the watchdog timer approach to transient fault tolerance is viewed as incomplete in itself, but useful as an option to be incorporated within the containment method.

References

[1] Glaser, R.E., and G.M. Masson, "Transient Upsets in Microprocessor Controllers," *Proceedings FTCS-11*, June 1981, pp. 165-167.

[2] Glaser, R.E., and G.M. Masson, "The Containment Set Approach to Upsets in Digital Systems," *IEEE Transactions on Computers*, July 1982.

[3] Schneider, F.B., and R.D. Schlichting, "Towards Fault Tolerant Process Control Software," *Proceedings FTCS-11*, June 1981, pp. 48-55.

[4] Intel Corp., *MCS-85 User's Manual*, Santa Clara, CA, January 1978.

[5] Dehnhardt, W., and V. Sorensen, "Unspecified 8085 op codes enhance programming," *Electronics*, January 18, 1979, pp. 144-145.

[6] Dennis, J.B., and E.C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, March 1966, Vol. 9, No. 3, pp. 143-155.

[7] Kraft, G.D., and W.N. Toy, *Microprogrammed Control and Reliable Design of Small Computers*, Englewood Cliffs, N.J., Prentice Hall, 1981, pp. 321-323.

[8] Ballard, D.R., "Designing Fail- Safe Microprocessor Systems," *Electronics*, Vol. 52, No. 1, January 4, 1979, pp. 139-143.

[9] Courtois, B., "Some Results about the Efficiency of Simple Mechanisms for the Detection of Microcomputer Malfunctions," *Proceedings FTCS-9*, June 1979, pp. 71-74.

[10] Courtois, B., "A Methodology for On-Line Testing of Microprocessors," *Proceedings FTCS-11*, June 1981, pp. 272-274.