Application Note

Using Software Techniques to Maximize Z8 MCU System Noise Immunity

AN003701-Z8X0400

ZILOG WORLDWIDE HEADQUARTERS • 910 E. HAMILTON AVENUE • CAMPBELL, CA 95008
TELEPHONE: 408.558.8500 • FAX: 408.558.8300 • WWW.ZILOG.COM

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Headquarters
910 E. Hamilton Avenue
Campbell, CA 95008
Telephone: 408.558.8500
Fax: 408.558.8300
www.ZiLOG.com

Information Integrity

Document Disclaimer

Table of Contents

Acknowledgements

Project Lead Engineer

Jon Veres

Application and Support Engineers

Denny Hopp, Oscar Miramontes, and J.R. Wang

System and Code Development

Jon Veres

Using Software Techniques to Maximize Z8 MCU System Noise Immunity

General Overview

Many articles are written about the hardware aspects of MCU-based system design. These articles cover topics such as PCB layout, component selection, and circuit protection devices. Unfortunately, the role software design can play to maximize noise immunity is often neglected. Good software design helps a system survive problems caused by noise that circumvents hardware protection schemes. This Application Note provides an introduction to a variety of software techniques that can provide a Z8 MCU-based system with improved noise immunity and reliability of operation.

Discussion

Fault-Avoidance and Fault-Tolerant Design

Hardware Design Methodologies

Many product designers think of fault avoidance and fault tolerance in terms of hardware system redundancies. The rationale is to prevent improper system actions when external and internal influences are encountered. External influences include supply, voltage, and frequency variations as well as EMI events (voltage surges, fast transients, electromagnetic radiation, and electrostatic discharge) and climatic conditions (short- and long-term temperature and humidity effects). Internal influences include systematic and random errors. Systematic or design errors, as they are more commonly called, occur because of complex product design. The errors are solved only through redesign efforts. Despite diligence in component selection and product construction, random errors and failures cannot be totally prevented. The system designer typically uses Failure Mode Effect Analysis (FMEA) or Fault Tree Analysis (FTA) techniques to identify deficiencies in end-product performance when various components randomly fail. These operational deficiencies are then addressed and corrected.

Extending Hardware Design Methodologies to Software

Typical hardware design methodologies can be extended to software. Elimination of systematic errors is a primary goal because of the MCU system software's necessarily unique nature. After a thorough analysis of the end-product performance requirements, there are several methods to use in the software design and test phases. These methods include:

- Writing structured code. Structured code includes dividing all system tasks into software blocks that can be tested individually. Each block must have a single entry point, a normal exit point, a fault exit point (if required), and a clear specification of data on which the block executes.
- Full-time system control. All software-controlled hardware functions must be in a known inactive state during program self-test, initialization, and fault-detected lock-out modes.
- Full documentation. Well-structured, thorough documentation greatly improves the prospects of eliminating systematic errors through individual and peer reviews, as well as ensuring the implemented functions fulfill the original design goals.
- Testing methodologies. To detect systematic errors, input and path testing techniques are used to exercise many module input conditions and program loops.

MCU system software can be designed for improved fault tolerance by use of on-line fault detection safeguards. These safeguards provide an interlock between system hardware and software operations. When faults are detected, the program determines the severity of the fault and takes the defined course of action. This action includes recording the error, prompting the end user, shutting down and locking out the system, as well as noting the speed at which corrective action is implemented.

The concept of fault tolerant software is promulgated by several standards agencies (for example, UL 1998 Table 7 parameters) for products that control potentially dangerous operations, such as a gas burner control. These standards address critical areas of operations that dictate the requirement for primary and secondary safeguard systems (hardware and software). The software methods discussed in this Application Note focus on the three MCU areas where errors occur or are most often induced: MCU memory, I/O, and program flow. Graceful recovery from a controlled system lock-out from any externally induced fault or random component failure is the goal of system design.

Implementing Fault Tolerance with the Z8

MCU Memory

The Z8 features internal program memory (ROM or One Time Programmable (OTP)) and register file RAM memory. A variety of program memory and RAM-checking methods are used to validate memory data and check for stuck bits as well as single- and double-bit errors. The techniques used include checksums, parity bits, CRC, and comparison of redundant memory segments. These memory tests are usually conducted and completed at power-up, but they also can be run continuously as part of the normal program scan. In the latter case, partial memory tests usually are conducted during each scan so that minimum MCU processing time is lost.

The sample Z8 program in the Technical Support section contains a program memory test routine. This routine computes and verifies a 16-bit ROM/OTP checksum. It provides an entire program memory check at power-up as well as a continuous background check (256 bytes/program scan). As seen from the sample code, the architecture of the Z8 MCU uses the Load Constant (LDC) instruction to make program memory testing easy. This instruction allows the user program direct access to memory and allows the memory to be operated on accordingly (also making the ROM/OTP look-up tables easy to use). Program memory tests can be conducted in Z8 MCUs even when the ROM/OTP Protect option is enabled. A detected and verified program memory fault is a *hard* or fatal system failure that terminates module operation.

The Technical Support Section also contains a sample Z8 register file RAM check routine. To test the register file, the RAM check routine writes and then verifies alternating byte patterns to all general purpose register locations. Thus, all address and data bit and byte combinations are exercised. When finished, the RAM check routine writes and verifies that all locations are cleared to zero. The RAM check is designed as a one-time test called during the system power-up initialization. The routine can be modified to perform partial register file checks continuously per program scan (as in the previous program memory test example). An on-going checksum test also can be computed on the entire register file with continuous updates for every data write. A simpler approach is to store all critical register file data in redundant locations. Then, as part of the system test procedures called each program scan, these memory locations are compared.

For example, assume there are 32 critical data parameters present in a user program. The data is stored in Z8 register file banks 1 and 2 (locations `10h` through `2Fh`). Fault-tolerant design techniques can be implemented by storing the complement of this data in another pair of register banks (mirror image data at register file banks 4 and 5 at locations `40h` through `5Fh`). After each program scan, the register file banks are compared for matching values. The Z8 instruction set supports this type of data storage and comparison through these instructions:

- Complement (COM)
- Exclusive-Or (XOR)
- Loop Count (Decrement and Jump if Not Zero, or DJNZ)

A sample bank checking code sequence for this example follows:

```
error rou =srp    #00h  ;work in register bank 0
                        ;define r5, r6, r7, r8 as local
                        ;variables for program data integrity
                        ;checking
ld r5, #10h             ;set-up primary data bank register pointer
ld       r6, #40h       ;set-up complimentary data bank register
                        ;pointer
ld       r7, #20h       ;initialize the byte compare counter (32)
                        ;set-up complete – now compare the primary
                        ;and complimentary redundant register
                        ;banks
loop:    ld r8, @r6     ;load the compliment register data
xor      r8, @r5        ;compare the compliment with the primary
                        ;data value
cp       r8, #0FFh      ;compare XOR result with FF
jp       nz, ram_fault  ;if nz, no data match--go to ram error
                        ;routine
djnz     r7, loop       ;repeat register data test loop for all 32
                        ;bytes
```

A detected and verified RAM error is usually a soft error. The recovery procedure validates that the RAM location in question is operating correctly (by performing RAM pattern testing) and then restores the data to the parameter's default or current state value.

MCU I/O

Despite hardware protection, MCU I/O is often subject to external influences. Software techniques such as input debouncing and I/O validity- and range-checking are available to ensure system I/O integrity. Fault-tolerance is improved by creating and maintaining a redundant and complementary I/O image table within the Z8 register file.

Software input debouncing is analogous to hardware input debouncing and keeps out noise-induced transients from being seen by the user program as actual input-state changes. There are different ways to implement input debouncing. The simplest technique is to count like-kind consecutive input states and, when the specified number is reached, interpret that input state as a valid. The problem with this method is that valid-state changes can be delayed adversely by repetitive noise conditions. An improved debounce method is to take a continuous signal-level-weighted average to determine the valid input state. This method assigns a debounce counter for each input port bit to be debounced. A low- and high-level threshold count is set for the user application, and the difference between the levels is the degree of hysteresis. An example of this type of input debounce method follows:

```
                        ;deb: subroutine called to debounce Z8
                        ;input port pin p25, report debounced pin
                        ;state to the port 2 I/O image register
                        ;
                        ;general purpose registers used:
                        ;TEMP: temp storage register
                        ;P25_CNT: debounce count register
                              ;
                              ;constants required:
                              ;P25_HI: user defined number of high
                              ;sample counts for a valid high
                              ;threshold
                              ;BIT5: 40h → binary 0010 0000
;
deb:    ld   TEMP, p2         ;read current state of p2 (do not
                              ;use boolean instruction on I/O
                              ;port)
        tm   TEMP, #BITS5     ;test p25 for a low level
        jr nz, deb_2          ;jp if p25 is high
        cp   P25_CNT, #00h    ;test for min debounce cnt value
                              ;(set to 0)
        jr   eq, deb_1
        dec  P25_CNT          ;dec hi level debounce counter
        ret
;
deb_1:  and  P2_IMAGE, #~BIT5 ;reset the p25 image bit/flag
        ret
;
deb_2:  cp   P25_CNT, #P25_HI ;test for max hi level debounce cnt
        jr   eq, deb_3
        inc  P25_CNT          ;inc switch hi level debounce
                              ;counter
        ret
;
deb_3:  or   P2_IMAGE, #BIT5  ;set the p25 image bit/flag
        ret
;
```

In the Z8, I/O port registers are mapped into locations `00h–03h` of the register file. Z8 fault tolerance is enhanced by constructing a redundant store of these registers. I/O image tables allow for redundancy checking on the debounced input states. The user program makes logical decisions based on this activity The input debounce counters (see the previous input debounce example) can be compared to the corresponding input port image table bit for logic state consensus. If a discrepancy is found, recovery options include changing the input to its reset default state.

Recovery options also provide for output state checking. Each Z8 bidirectional I/O port features an input register and buffer, an output register and buffer, and associated control logic. Writing to the I/O port register stores data in the port's output register. This feature allows output port data to be initialized prior to the I/O pins being configured as outputs. Reading a port register causes the data on the external I/O port pins to be read whether defined as inputs or outputs. The output data read in is compared to the output data of the I/O image table. If the output data matches, the outputs are properly represented on the external interface and there are no shorts or stuck pins. This type of output validation checking is performed only on outputs configured for push/pull active drive.

Finally, I/O image tables allow the safe use of Z8 Boolean instructions (TM, TCM, AND, OR, XOR, COM). Z8 Boolean instructions operate on a Read–Modify–Write basis. If the user program directly operates on an I/O port with a Boolean instruction, there is the potential danger of a noise pulse corrupting a bidirectional I/O bit. Therefore, it is safer to perform all user program I/O modifications on the I/O port image table registers while always using the Load (LD) instruction to act on the actual I/O ports.

MCU Program Flow

Normal MCU program flow also may be interrupted by external influences despite hardware protection. Typical fault modes here are the altering of the MCU program counter (causing the software to jump to random locations) and a noisy reset line (which locks-up MCU operation). There are several hardware and software techniques embedded in the design that can help a system recover from these faults.

Hardware Methods

The Z8 CCP MCU family provides on-chip hardware resources to aid in EMI upset recovery. A primary feature is the elimination of the external reset line in favor of internal hardware low voltage/brown-out detection and reset control. This feature eliminates the possibility of improper MCU operation or lock up because of noisy or glitching reset input which would violate MCU reset input timing specifications. On-chip low-voltage detection guarantees the Z8 is in a known operating state (RUN, HALT, STOP, or RESET) during V_{CC} transients. This feature eliminates the requirement for a reset input and its associated external support and conditioning circuitry on the Z8. When in reset mode, all Z8 bidirectional I/O are configured as high-impedance inputs so no output loads can be driven (autolatches disabled). This condition is the same I/O condition that is present after reset release during power-up initialization. Z8 outputs are enabled only after the I/O direction bits are configured in the I/O port mode control registers.

The Z8's on-chip timer/counters are used to validate MCU system timing. A Z8 timer is used to periodically measure the time duration of a known applied external signal such as a fixed-input clock frequency, 60-Hz AC line (zero cross detect), and so on. When the external signal is in a valid range, the user program knows the MCU program time base (clock source) is within design specification. Likewise, a system dependent on a 60-Hz zero cross input can use the internal MCU time base to validate the external timing signal. For example, assume a timer/display module based on an external 60-Hz signal. The MCU monitors the timing of zero cross input. If present, normal module operation is continued. If missing, the module could enter a diagnostic low-power mode. Low-power mode consists of turning off all power-consuming displays and waiting for the proper zero cross resumption. In this case, module operation might sustain multiple second *brown-outs* without affecting continued operation (transparent continuous operation to the end user).

Also available on the Z8 CCP MCU is an on-chip watch-dog timer (WDT) circuit. A WDT is a hardware circuit that counts input pulses. If it counts to a predetermined maximum number of counts, the WDT generates a hardware timeout signal that can be used as a fault lock-out signal or master system reset. When operating normally, the user program retriggers, or resets, the WDT count to its initial value before the count expires and generates the system reset. Thus, there is a hardware safeguard on system software operation.

The most reliable on-chip MCU WDT contains a number of operational properties. The WDT clock source must be reliable and independent of MCU system hardware. The WDT should offer timing flexibility to match the user application by having a time-out period consistent with the process being controlled (not too fast or too slow). Because the WDT is a hardware safeguard on controlled software action, the WDT must remain as independent of the software as possible. Thus, the WDT should be initiated by application of system power and armed independently of software that may not run properly at start-up. Similarly, software should not be allowed to disable an operating WDT. *Kicking* or retriggering the WDT (resetting the timer count to its initial value) should be through a unique software command. Finally, the WDT time-out should generate a *safe* system reset condition.

Some applications require the use of an external WDT to force safety critical hardware to an *off* state. A well-designed on-chip WDT suffices for many applications. The Z8 CCP MCU's internal WDT features provide maximum user flexibility and system protection. Those features include:

- Internal clock source—WDT operation and reset are independent of the MCU clock source.
- Automatic WDT start-up at power-on-reset (feature enabled via a ROM mask or an OTP programming option bit).
- When enabled, the WDT cannot be disabled by software.
- Unique WDT refresh Op Code.

- Software programmable modes of operation via the access protected WDTMR register:
  - WDT enabled in any/all MCU operating modes: RUN, HALT, STOP
  - Programmable WDT time-out periods (5 msecs to 100 msecs)

In the Z8 CCP MCU, WDT operation options are controlled by the Watch-Dog Timer Mode Register (WDTMR). This register is written only by the user program during the first 64 system clock cycles after a POR, WDT RESET, or STOP-mode recovery. This access protection prevents rogue writes that can occur with program counter corruption. This access protection also controls time-out period selection and watch-dog timer activity in the various MCU operating modes.

Although well designed, the Z8's on-chip WDT provides maximum protection only if properly used. The WDT is retriggered by software execution of the WDT instruction (`5Fh` Op Code). For maximum effectiveness, the user program should retrigger the WDT at only one point in the main program. The retrigger operation is usually performed after all software system checks are validated. This concept is discussed in the Software Methods section that follows.

Finally, all Z8 CCP MCU I/O pins feature internal input protection diodes to the high- and low-side rails. This feature provides effective device protection against voltage transients with the addition of current limiting resistors.

Software Methods:

In addition to the hardware safeguards on program flow, user software can actively detect signs of a system fault/upset. Software for detection of MCU memory and I/O faults has been reviewed. Similar software techniques prevent and detect illegal program flow. The simplest prevention method is to fill any unused ROM program memory with the Z8's NOP Op Code (`FFh`) up to the most recent three program bytes where a jump to START instruction is located.

Example:

```
...                            ;last user program byte
...                            ;unused space filled with NOPs
...
nop
nop
jp start                       ;periodic jp to start instruction
...                            ;unused space filled with nop’s
...
...
nop
nop
jp start                       ;end of Z8 program memory space
```

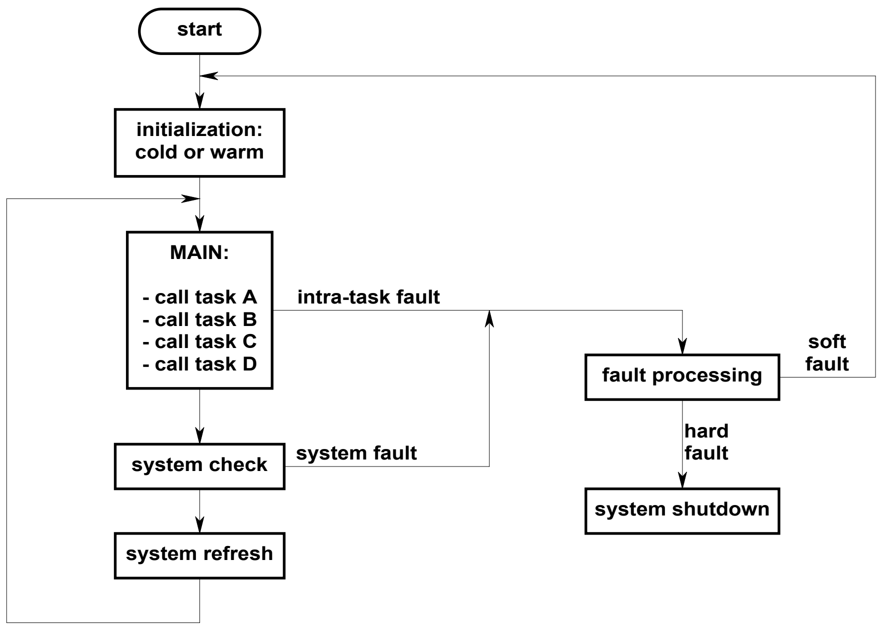
In this example, START is the label for program address `000Ch` (starting program address location for all Z8 MCUs). If the program counter (PC) is corrupted and execution is redirected to the unused memory area, the Z8's PC would eventually resynchronize on the NOP instructions and jump to the start location (or any other fault-handling code address selected by the programmer).

Similarly, all unused Z8 interrupt sources must have their jump vectors pointed to a *bad interrupt* system-recovery routine. This routine flushes the interrupt condition by restoring the flag register and stack pointer, and directing the program counter to continue at the user's desired location.

Another prevention measure examines the user program hex file and eliminates all Z8 MCU operating mode Op Codes that appear as part of the normal program. The critical Op Codes are for the HALT (`7Fh`), STOP (`6Fh`), and WDT (`5Fh`) instructions. For example, the instruction LD (R5 `6Fh`) loads the immediate value of `6Fh` into working register 5. This operation provides a machine code sequence of 5C 6F. If a program counter disruption occurs, the potential exists for the PC to start execution with the 6F Op Code, and thus execute a STOP instruction. For this reason, these critical Op Codes exist only in one place: the actual routine where a power-down mode is entered or the watch-dog timer is kicked. These power-down routines incorporate software checks to validate their execution and guardband the Z8's WDT.

There are several software methods to detect program flow disruption. Consider the simplified user program flow in Figure 1.

Figure 1. Simplified Z8 User Program Flow



Proper program flow executes tasks in an A, B, C, D order. By using unique, pre-assigned sequence numbers for each subroutine, system software validates proper program flow. In this example, the main program sets the initial sequence number, then calls Task A. Task A, when starting, running, and/or concluding execution, matches the sequence number to the supervisory sequence number. If the sequence numbers do not match, Task A directs program execution to the fault handler. This process continues as the main program sets the predetermined sequence numbers for each succeeding task called.

Additional fault checking is carried out at the task level. Each task monitors its passed input parameters for validity. Time-slot monitoring is a technique that uses the Z8 MCU timer resource to measure the duration of each task's execution. Execution time outside the acceptable minimum/maximum timing window is signaled as an operating fault. Another method monitors intra-task logical flow (program sequencing, discussed previously, is an inter-task logical flow monitoring technique). Checkpoint flags are set up inside each task block to ensure that all critical functions inside a task are executed. At task end, the checkpoint flags, a unique one for each task function, are checked to see that each critical function ran one time, and only one time. A simplified and less precise implementation makes use of a flow-check counter. When each critical task function is completed, the flow-check counter is incremented. At the end of the task, the flow-check counter is tested for the proper count. If checkpoint flags are not set correctly or flow-check counts do not match, the nonmatch indicates a program flow in the task was upset and critical functions were either not executed or were executed repeatedly. Program action is then directed to the fault-handler software.

Note: These same methods are used in interrupt service routines to ensure that the routines are not being called too frequently (or too infrequently), relative to main program operation.

To continue with this example, when the main loop completes execution, the system check and refresh routine runs. This subroutine performs global system checks such as the background program memory check, critical redundant RAM checks, system time base check, and final sequence code check. If all global checks pass, the system is in control and system refresh begins. Refresh includes updating and rewriting the I/O ports from the I/O image table, the I/O port mode control registers, system control registers (interrupt and stack facilities), and timer control registers. The purpose of the refresh is to ensure that any noise-corrupted system control registers are reinitialized to the required state. Finally, the WDT instruction is executed, the only time the WDT instruction is executed.

When any of these software procedures detects a fault, the user program jumps to the fault-handling routine. This routine determines the fault root cause and takes the user-defined action. The action taken depends on whether the error is considered fatal (results in system lock-out) or recoverable (try a system restart). The system designer decides what a fatal and recoverable error is for each end-product. The fault handler makes use of the sequence code and checkpoint flags determine the actual root cause failure mode. Thus, sequence numbers and checkpoint flags serve a secondary role as fault *tracing codes* that indicate when proper system operation terminated.

When the user program vectors back to the program start location (whether by WDT, bad interrupt vector, or by action of the fault handler), the user's initialization sequence must account for this type of restart. The restart is accomplished by having the system initialization code determine if a warm or cold start condition exists. A cold start is defined as an initial power-up sequence. For this case, the complete system check and initialization to system default state must be performed. A warm start is defined as a system that was running, but for some reason (system upset/fault detection/WDT time-out), is attempting a restart. Instead of defaulting to the cold start reset sequence, the system checks to see if it can carry on program activity from the point of the upset (or as close as possible to it). Recovery can be as simple as checking the RAM signature bytes (fixed register file locations programmed to unique data values on a cold start), or to more sophisticated RAM checks. If critical system RAM is intact, program execution continues at the main program without reinitializing to the cold start I/O default conditions.

In the example in Figure 1, the redundant register banks holding all critical program and I/O data are tested to determine a warm or cold start recovery. If RAM data checks out as valid, system operation proceeds as a warm start. In this case, there is no glitching of I/O as the ports remain in the most recent state condition. To the product's end user, the system glitch goes unnoticed, as system operation is on-going.

Summary

This Application Note focuses on available software methods to enhance a Z8 MCU-based system's immunity to EMI upset. Topics discussed range from improved documentation and testing practices to specific techniques for preventing and detecting faults in MCU memory, I/O, and program flow. These methods were developed to meet the standards required for safety-critical products. These same techniques can be used, as MCU resources allow, in any embedded control design to enhance the end product's reliability.

Technical Support

Source Code

```
************************************************************************
*     Module Name:   FT_SFWR.ASM
*     Version:       V1.0
*     Copyright:     ZiLOG (c)1999
*     Date:          August 23, 1999
*     Created by:          Jon Veres - ZiLOG Ohio
*     Compiler:      ZDS V2.11
*     Description:   Demonstration program to outline fault tolerant
*                    software techniques including program/RAM memory
*                    checking and task flow control. Target Z8 device
*                    for this example is the Z86x04.
************************************************************************
;
      globals on
;
;     tab size = 6
;
; User application definitions
;
ROM_LO      equ   0FFh               ; stored ROM checksum Low byte value
ROM_HI      equ   65h                ; stored ROM checksum High byte value
ROM_SIZE    equ   4                  ; 256 byte pages in selected Z8 MCU
SIG1        equ   00h                ; RAM cold/warm start signature bytes
SIG2        equ   55h
;
;
RAM_VAL     equ   0FFh               ; RAM test pattern byte (GPR)
RAM_PTR     equ   0FEh               ; RAM test pointer (SPL)
;
RAM_MAX     equ   7Fh                ; top of GP reg file for Z8 used
RAM_MIN     equ   3                  ; bottom of GP reg file for Z8 used
PATTERN     equ   55h                ; selected RAM test pattern byte
;
;     section definitions
;
; Bank 1 working register definitions (ROM test control registers)
;
rom_ptr     equ   rr0                ; wr0 pr - 16-bit ROM address ptr
rom_ptr_hi  equ   r0                 ; wr0 - ROM addr ptr high
rom_ptr_lo  equ   r1                 ; wr1 - ROM addr ptr low
byte_cnt    equ   r2                 ; wr2 - ROM check byte counter
rom_pass    equ   r3                 ; wr3 - on-line ROM check counter
rom_char    equ   r4                 ; wr4 - ROM check summing reg
chksum_lo   equ   r5                 ; wr5 - computed ROM chksum low
chksum_hi   equ   r6                 ; wr6 - computed ROM chksum high
;
      define bank2_data, space=rfile, org=20h ; mapped to 20-2F
;
      segment bank2_data
;
RAM_SIG     ds    1                  ; generic reg file definitions
RAM_SIG1    ds    1
RAM_SIG2    ds    1
```

```
RAM_SIG3    ds    1
FLOW_CNT    ds    1
;
;
**************************************************************************
*     Interrupt Vectors
**************************************************************************

      vector      reset = begin
      vector      irq0  = IRQ0
      vector      irq1  = IRQ1
      vector      irq2  = IRQ2
      vector      irq3  = IRQ3
      vector      irq4  = IRQ4
      vector      irq5  = IRQ5
;
;
      segment code
;
; start of program
;
begin:      di                          ; start at 000Ch, disable int
;
; Configure ports and port control registers
;
            srp   #00h                  ; enable working reg bank 0

; Test for POR (cold) start or EMC/static reset (warm) start by looking
; for the proper RAM signature.
;
            cp    RAM_SIG, #SIG1        ; proper RAM signature pattern?
            jr    ne, cold
            cp    RAM_SIG1, #SIG2
            jr    ne, cold
            cp    RAM_SIG2, #~SIG1
            jr    ne, cold
            cp    RAM_SIG3, #~SIG2
            jr    eq, warm              ; RAM signature pattern test
                                        ; passed, jp to warm start
                                        ; location
;
; POR detected -- perform cold start
;
cold:       ld    p01m, #4Dh            ; 4D = int stack only, all
                                        ; I/O as inputs
;
; Perform a RAM check on the entire general purpose (GP)Z8 register file.
; The test consists of writing all GP registers with an alternating byte
; test pattern, verifying the pattern, then repeating the entire procedure
; writing and reading the complemented pattern.
;
; SFR R254 - SPL: defined as RAM_VAL - holds the test pattern byte
; SFR R255 - GPR: defined as RAM_PTR - reg file address pointer
;
; User defines the following constants for the application:
;
; RAM_MAX: highest GP reg file address available for the Z8 selected
; RAM_MIN: lowest GP reg file address available for the z8 selected
; PATTERN: value of the user defined test pattern byte
;
            ld    RAM_VAL, #PATTERN       ; init RAM check pattern value
ram_wr:     ld    RAM_PTR, #RAM_MAX       ; init RAM ptr to top
ram_wr_1:   ld    @RAM_PTR, RAM_VAL       ; write out RAM pattern
            dec   RAM_PTR
            cp    RAM_PTR, #RAM_MIN       ; all GP reg bytes written?
            jr    eq, ram_rd
            com   RAM_VAL                 ; complement RAM pattern value
            jr    ram_wr_1
;
; General purpose reg file write with alternating test/compliment test
; bytes completed, read back and verify each byte value is correct.
;
ram_rd:     ld    RAM_PTR, #RAM_MAX       ; set RAM ptr to the top
ram_rd_1:   com   RAM_VAL                 ; init RAM test byte value
            cp    RAM_VAL, @RAM_PTR       ; compare RAM patterns
            jp    ne, ram_err
            dec   RAM_PTR                 ; point to next reg file byte
            cp    RAM_PTR, #RAM_MIN       ; all GP bytes checked?
            jr    ne, ram_rd_1
;
pass_2:     cp    RAM_VAL, #~PATTERN      ; 1st or 2nd RAM test pass
            jr    ne, ram_clr             ; if 2nd, go to clear ram
            jr    ram_wr                  ; if 1st, start 2nd test pass
;
; General purpose register file testing been successfully completed,
; clear/verify clear all general purpose bytes.
;
ram_clr:    ld    RAM_PTR, #RAM_MAX       ; set ptr to top of reg file
clr_it:     clr   @RAM_PTR                ; zero out all gp reg file
            cp    @RAM_PTR, #00h          ; verify RAM byte cleared
            jp    nz, ram_err
            dec   RAM_PTR                 ; point to next reg file byte
            cp    RAM_PTR, #RAM_MIN       ; all GP bytes cleared?
            jr    ne, clr_it
;
; Register file testing is complete, proceed to check the program memory.
;
            ld    spl, #RAM_MAX+1         ; init stack pointer
            call  test_rom                ; validate program memory
;
; Program and register file memory test OK, now initialize the outputs and
; RAM.
;
            ld    p0, #07h                ; write the output buffers
            ld    p2, #01h
            ld    RAM_SIG, #SIG1          ; set-up RAM signature test
            ld    RAM_SIG1, #SIG2
            ld    RAM_SIG2, #~SIG1
            ld    RAM_SIG3, #~SIG2
;
warm:       push  rp
            srp   #10h                    ; reinit the on-line ROM test
            call  init_rom                ; regs after a warm start reset

            pop   rp
;
```

```
; Ready for main program - first refresh the system control registers.
;
main:       ld    p01m, #04h              ; 04 = int stack, p0 = outputs
            ld    p3m, #03h               ; p3.1-3.3= an in, p2= push/pull
            ld    p2m, #0feh              ; p20 = out, p21-27 = in
            ld    spl, #RAM_MAX+1         ; init stack pointer
            clr   imr                     ; mask off all interrupt sources
            srp   #00h                    ; enable wr bank 0
            clr   FLOW_CNT                ; init program flow counter
;
; Refresh section of main is complete -- now run the main user program.
;
            call  task_A                  ; typical user program here
            cp    FLOW_CNT, #01h          ; proper flow check count?
            jp    ne, prgm_err            ; if flow cnt wrong - go to prgm
                                          ; fault routine
            call  task_B
            cp    FLOW_CNT, #02h          ; proper flow check count?
            jp    ne, prgm_err
            call  task_C
            cp    FLOW_CNT, #03h          ; proper flow check count?
            jp    ne, prgm_err
;
; The user program has run correctly, now run the on-line system
; diagnostics routines before starting the next program scan.
;
            cp    RAM_SIG, #SIG1          ; proper RAM signature pattern?
            jp    ne, ram_err
            cp    RAM_SIG1, #SIG2
            jp    ne, ram_err
            cp    RAM_SIG2, #~SIG1
            jp    ne, ram_err
            cp    RAM_SIG3, #~SIG2
            jp    ne, ram_err             ; RAM sig pattern test passed,
            call  tst_rom                 ; on-line ROM test (256 bytes/
                                          ; prgm scan)
;
            nop                           ; all other system diagnostics/
            nop                           ; refresh routines are called
            nop                           ; before jumping to start the
            nop                           ; next user program scan
;
            jr    main
;
;
*************************************************************************
*     test_rom:   Performs the cold start complete ROM check and verifies
*                 the computed 16-bit checksum with the stored checksum.
*                 If checksums do not match, program execution is directed
*                 to the ROM error routine. Otherwise the on-line ROM
*                 check registers are initialized and normal program
*                 execution continues. All ROM tests use the predefined
*                 registers of wr bank 1.
*************************************************************************
;
test_rom:   push  rp                      ; save current system rp
            srp   #10h                    ; select ROM check wr bank
            call  init_rom                ; init reg bank 1 for ROM check
```

```
test_1:     call  rom_chk                ; computes chksum for 256 bytes
            djnz  rom_pass, test_1       ; all ROM blocks in Z8 checked?
            call  chk_sum                ; if yes, perform chksum test
            pop   rp                     ; ROM ok - restore rp
            ret
;
;
*************************************************************************
*     tst_rom:    Performs the on-line running ROM check. This ROM check
*                 uses the predefined registers of wr bank 1 and checks
*                 256 byte ROM blocks per call/program scan. When all
*                 blocks have been summed, the stored 16-bit checksum is
*                 compared to the computed value. If checksums do not
*                 match, program execution is directed to the ROM error
*                 routine. Otherwise the on-line ROM check registers are
*                 initialized and normal program execution continues.
*************************************************************************
;
tst_rom:    push  rp                     ; save current system rp
            srp   #10h                   ; select ROM check wr bank
            call  rom_chk                ; compute chksum for 256 bytes
            djnz  rom_pass, tst_1        ; all ROM blocks in Z8 checked?
            call  chk_sum                ; if yes, perform chksum test
tst_1:      pop   rp                     ; ROM ok - restore rp
            ret
;
*************************************************************************
*     init_rom:   Initialize the reserved wr in bank 1 for the cold start
*                 and run time on-line ROM checks.
*************************************************************************
;
init_rom:   ld    rom_pass, #ROM_SIZE    ; counts number of 256 byte ROM
                                         ; checks 1K ROM = 4 ROM PASSES
            clr   chksum_lo              ; init running ROM check regs
            clr   chksum_hi
            clr   rom_ptr_lo
            clr   rom_ptr_hi
            clr   byte_cnt
            ret
;
*************************************************************************
*     rom_chk:    Computes the running checksum in 256 byte blocks -
*               computed value stored in chksum_lo and chksum_hi.
*************************************************************************
;
rom_chk:    ldc   rom_char, @rom_ptr     ; load ROM byte to the reg file
            add   chksum_lo, rom_char    ; compute the running checksum
            adc   chksum_hi, #00h        ; fix add carry - 16-bit result
rom_1:      incw  rom_ptr                ; do next ROM byte
            djnz  byte_cnt, rom_chk      ; end of 256 byte ROM block?
            ret
;
*************************************************************************
*     chk_sum:    Compares the computed ROM checksum to the stored value.
*                 A failed checksum match causes program execution to jump
*                 to the ROM fault handler.
*************************************************************************
;
```

```
chk_sum:    cp    chksum_lo, #ROM_LO        ; all ROM tested, test checksum
            jp    ne, rom_err               ; bad chksum is fatal error
            cp    chksum_hi, #ROM_HI
            jp    ne, rom_err               ; bad checksum is a fatal error
            call  init_rom                  ; chksum good, set-up for run
            ret                             ; time continuous chksum
;
;
***************************************************************************
*     task X:     Typical user program tasks.
***************************************************************************

task_A:     nop                             ; user task A here
            inc   FLOW_CNT                  ; task A complete, inc flow cnt
            ret
;
task_B:     nop                             ; user task B here
            inc   FLOW_CNT                  ; task B complete, inc flow cnt
            ret
;
task_C:     nop                             ; user task C here
            inc   FLOW_CNT                  ; task C complete, inc flow cnt
            ret
;
;
***************************************************************************
*     rom_err:    User defined action for software detected ROM fault.
***************************************************************************
;
rom_err:    jr    rom_err                   ; user defined ROM fault handler
;
;
***************************************************************************
*     ram_err:    User defined action for software detected RAM fault.
***************************************************************************
;
ram_err:    jr    ram_err                   ; user defined RAM fault handler
;
;
***************************************************************************
*     prgm_err:   User defined action for software detected program flow
*                 error.
***************************************************************************
;
prgm_err:   jr    prgm_err                  ; user defined prgm flow error
;
;
***************************************************************************
*     Interrupt Service
***************************************************************************
;
      IRQ0:
      IRQ1:
      IRQ2:
      IRQ3:
      IRQ4:
      IRQ5:
            jp    begin
;
;
************************************************************************
*     Checksum Add Byte value declaration for checksum computation.
*
*     ROM_HI and ROM_LO are the stored 16-bit program checksum. The
*     "Checksum Add Byte" is used to make the ROM_LO byte equal to FFh.
*     To compute the stored checksum, the Checksum Add Byte should be
*     initialized = 00h. ROM_LO and ROM_HI should be initialized to 0FFh.
*
*     Run this ROM check program and compute the initial checksum test
*     result (set a breakpoint at the CHK_SUM routine label and read
*     the value of chksum_hi). The value computed for chksum_hi is then
*     inserted as the value of ROM_HI. The checksum test is then repeated
*     and the checksum add byte's final value is the difference needed to
*     make the chksum_lo byte value equal to FFh once the ROM_HI byte is
*     set (FF - final computed chksum_lo = checksum add byte).
*
*     Note: User must insure unused memory is set to the proper state prior
*     to running the checksum tests above.  For the Z8, set unused program
*     memory space to FFh (NOP instruction).
************************************************************************
;
            db    4Eh                     ; add byte to make ROM_LO = FFh
;
      .END
```

Flowcharts

The flowcharts on the next three pages illustrate the program flow of the `FT_SWR.asm` module.

Figure 2. FT_SWR.ASM Module Flow—Chart 1 of 3

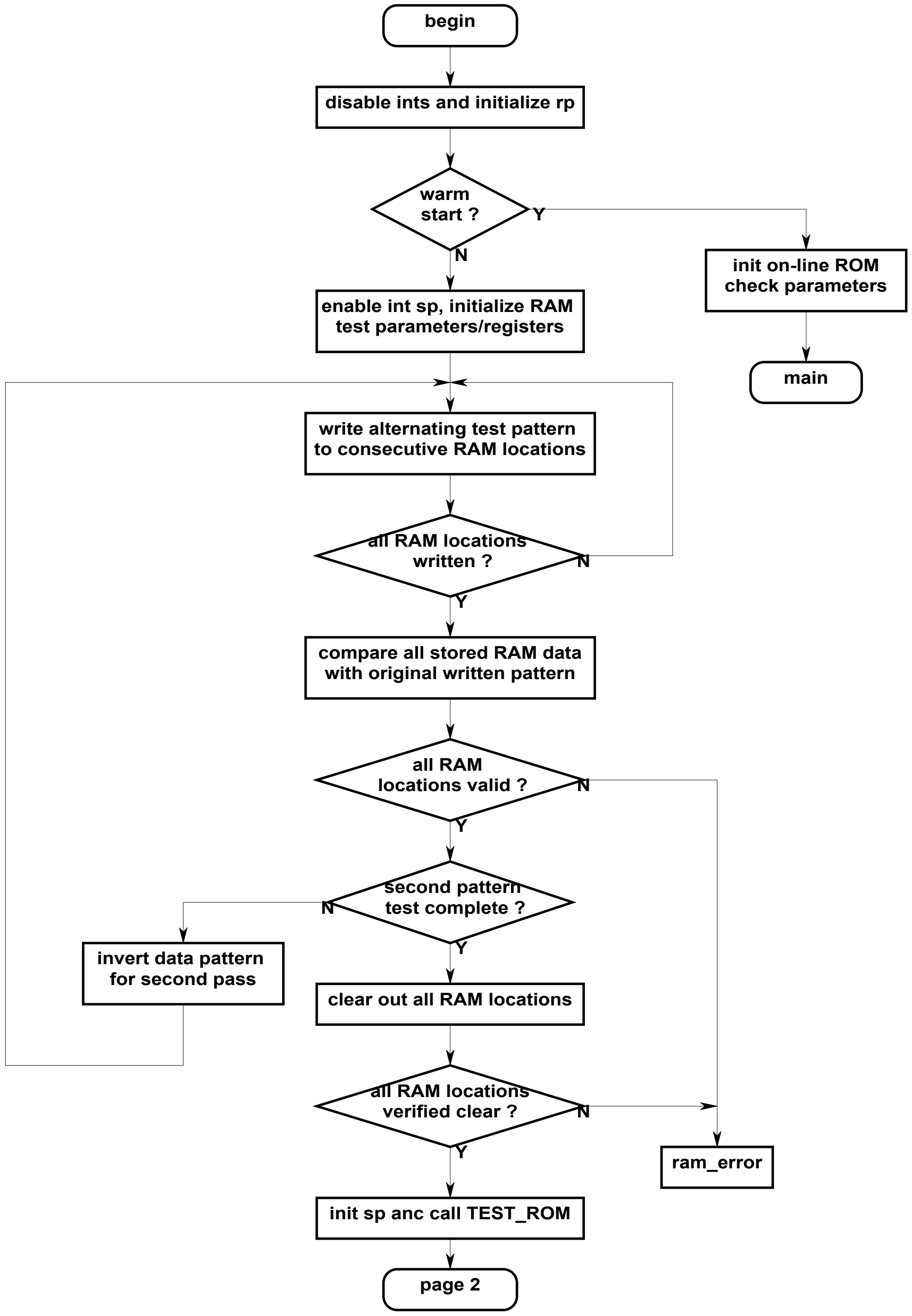

begin
disable ints and initialize rp
warm start ?
Y
N
init on-line ROM check parameters
main
enable int sp, initialize RAM test parameters/registers
write alternating test pattern to consecutive RAM locations
all RAM locations written ?
N
Y
compare all stored RAM data with original written pattern
all RAM locations valid ?
N
Y
second pattern test complete ?
N
Y
invert data pattern for second pass
clear out all RAM locations
all RAM locations verified clear ?
N
Y
ram_error
init sp anc call TEST_ROM
page 2

Figure 3. FT_SWR.ASM Module Flow—Chart 2 of 3

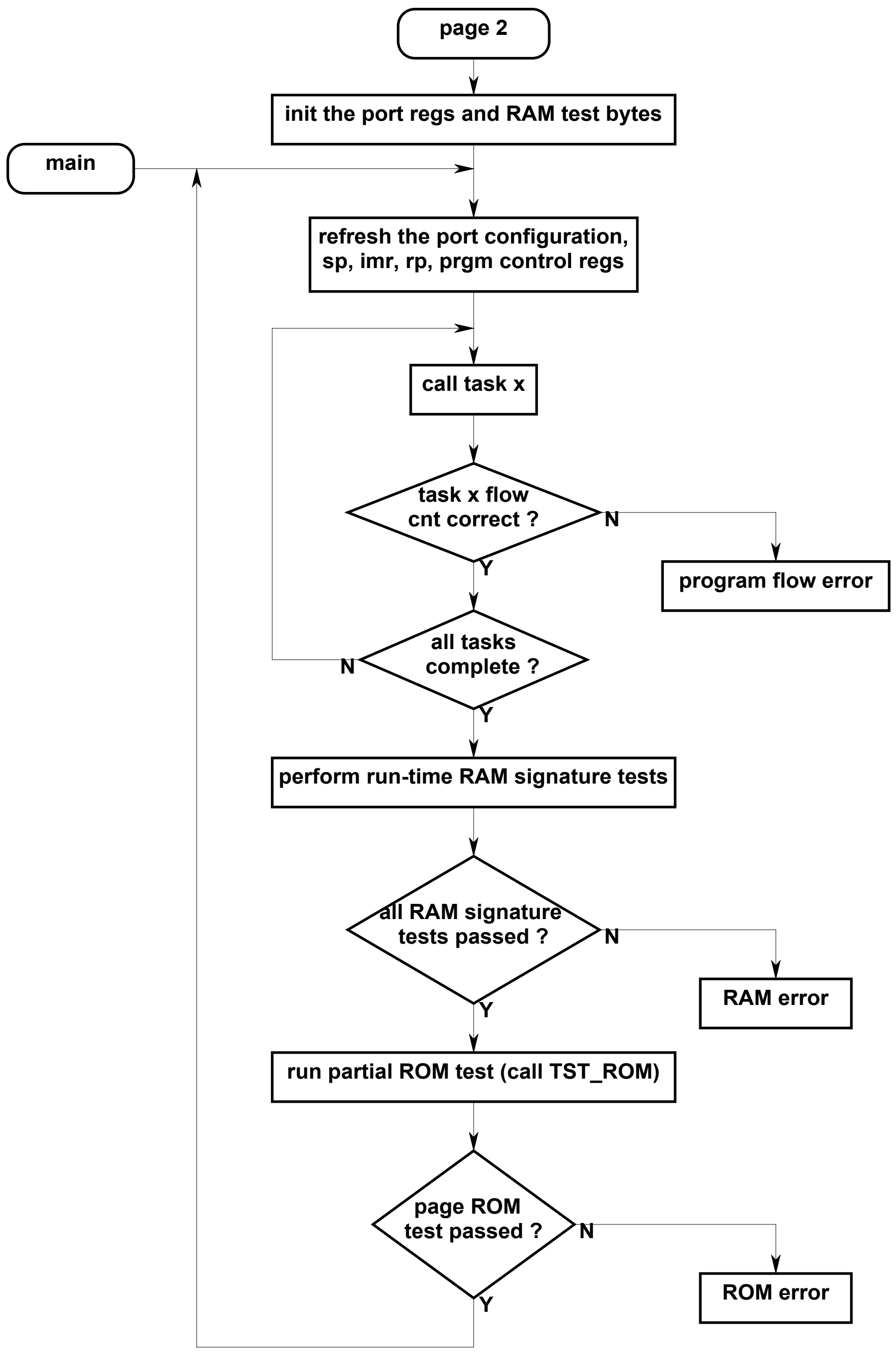
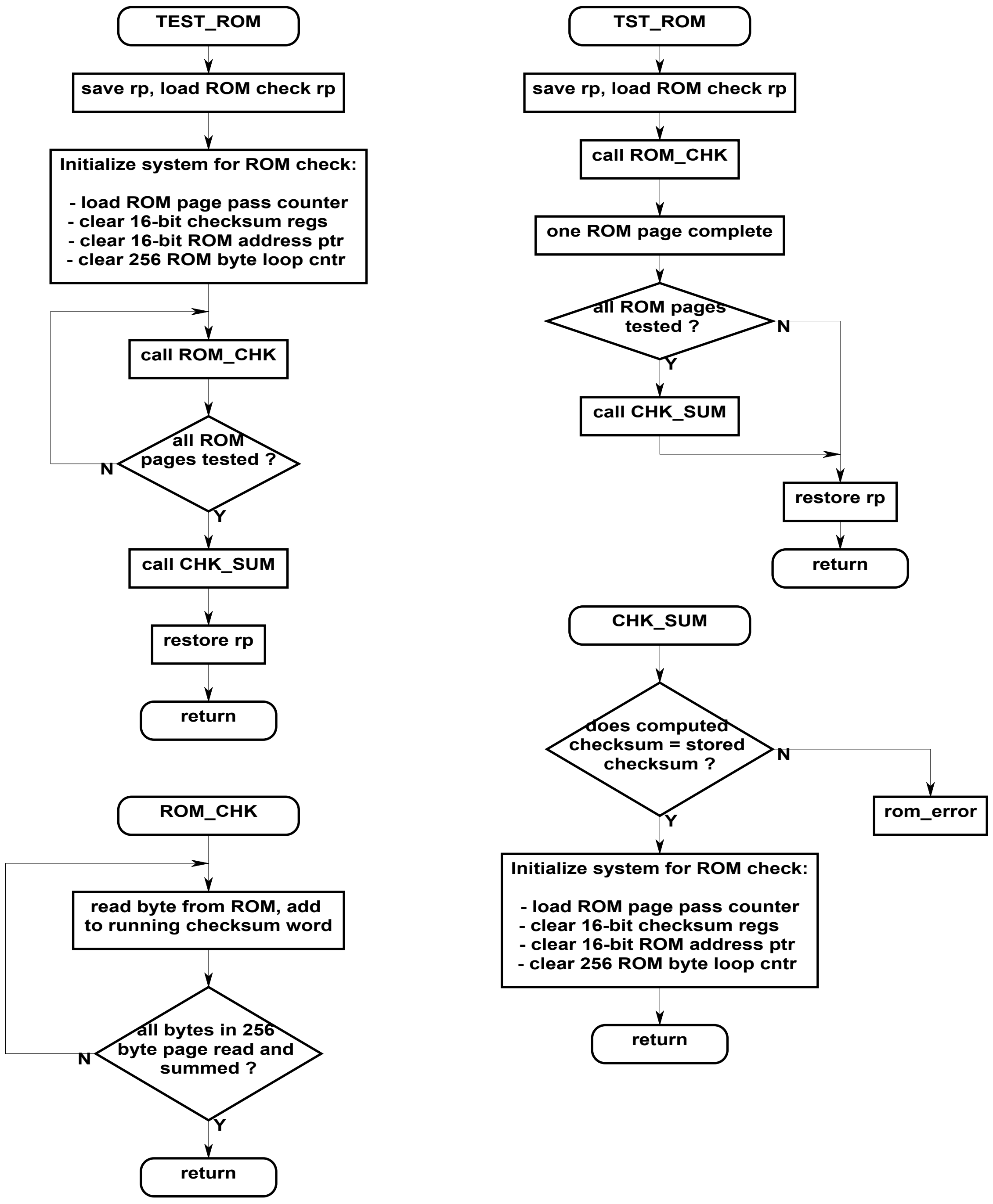

page 2
init the port regs and RAM test bytes
main
refresh the port configuration, sp, imr, rp, prgm control regs
call task x
task x flow cnt correct ?
N
program flow error
Y
all tasks complete ?
N
Y
perform run-time RAM signature tests
all RAM signature tests passed ?
N
RAM error
Y
run partial ROM test (call TST_ROM)
page ROM test passed ?
N
ROM error
Y

Figure 4. FT_SWR.ASM Module Flow—Chart 3 of 3


TEST_ROM
save rp, load ROM check rp
Initialize system for ROM check:
- load ROM page pass counter
- clear 16-bit checksum regs
- clear 16-bit ROM address ptr
- clear 256 ROM byte loop cntr
call ROM_CHK
all ROM pages tested ?
N
Y
call CHK_SUM
restore rp
return
ROM_CHK
read byte from ROM, add to running checksum word
all bytes in 256 byte page read and summed ?
N
Y
return
TST_ROM
save rp, load ROM check rp
call ROM_CHK
one ROM page complete
all ROM pages tested ?
N
Y
call CHK_SUM
restore rp
return
CHK_SUM
does computed checksum = stored checksum ?
N
Y
rom_error
Initialize system for ROM check:
- load ROM page pass counter
- clear 16-bit checksum regs
- clear 16-bit ROM address ptr
- clear 256 ROM byte loop cntr
return

Test Procedure

Equipment Used

Testing the sample Z8 program requires the following equipment:

- PC with Windows 95/98/NT and ZDS 2.11 installed
- Z86CCP01ZEM
- Z86CCP00ZAC (PC and power cables only)
- 8V power supply

General Test Setup and Execution

Four files are included in the `FT_SFWR.zip` file.

• `FT_SFWR.doc`	(this Application Note)
• `FT_SFWR.zws`	(ZDS 2.11 project file)
• `FT_SFWR.asm`	(sample FT software program source file)
• `FT_SFWR_TEST.soc`	(reviewer test document)

The testing was performed with ZDS 2.11 and the Z86C04 as the target chip. No target cable or target board is required.

ZDS 2.11 is used to assemble the source program (FT_SFWR.asm) and monitor the Z86C04 target program and register file memory windows. Program memory sizes (and target chips) from 512 to 16KB were checked for proper checksum test results. The ZCONVERT utility verified checksum computations over the program memory sizes tested. The RAM tests were verified on a maximum RAM size of 32, 64, 128, and 256 bytes.

Test Results

All program memory and RAM memory tests functioned correctly over the sizes tested. The program runs at all speed ranges (no frequency dependencies).

References

- Handbook of EU EMC Compliance, Compliance Design, Inc., 1995 edition
- UL 1998 Standard for Safety Related Software, Underwriters Laboratories
- Embedded Systems Programming, *Watch-Dog Timer Techniques*, April, 1995