MOTOROLA
Microprocessor and Memory Technologies Group

Order this document by
AN1264/D

AN1264

Application Note
JTAG Flash Memory Programmer

By James Gilbert, High Performance Embedded Systems, East Kilbride.

During the manufacturing process for the production of embedded processor systems, there is no firmware present in the on-board memory. While some manufacturers get the required software masked into ROM during the device fabrication, this is commonly expensive and certainly more restrictive in terms of new software releases. A more flexible approach is for the system manufacturer to upload the system software during their production cycle. To do so, means the system hardware must have a communication interface build into it. With pcb space at a premium, a serial interface is the preferred choice.

Some processor families offer a solution through their in-built serial debug interface. Examples are the CPU32, CPU32+ based MC683xx family, the embedded Powerpc MPC5xx/MPC8xx families and the MCF52xx Coldfire family, which utilise a background debug interface to allow a pc/sun host to remotely debug and control the processor and thereby the system. This is great in the flexibility and production time download stakes, but different processor manufacturers adopt their own particular serial implementation.

An alternative which is often overlooked is the JTAG interface. Many manufacturers consider IEEE1149 JTAG Test Access Port as mandatory for system and integrated circuit integrity testing using automatic test equipment. As such it is already present in the system, and so is an ideal candidate for a firmware loading mechanism. The download speeds possible using JTAG may prohibit its practicality for downloading the complete system software, but certainly it is ideally suited to download a small bootloader program into Flash EPROM to control the remainder of the system download. Another key application area the technique supports are field upgrades where new software releases are loaded onto an existing customer system.

This document describes a JTAG Flash EPROM programmer designed to run on an IBM compatible PC. While the programmer was tested using MC68307 and MC68306 processor systems, simply changing the two scan register input files makes the programmer equally applicable to any (well at least many) other processor with a JTAG IEEE1149.1 compliant port.

OVERVIEW OF IMPLEMENTATION

An overview of the JTAG Flash programmer concept is provided in Figure 1, and a full listing of the software is available in Listing 1. The programmer is written in 'C' and compiled using the Turbo C/C++ compiler (Version 3.0) to give an executable file (jtagprog.exe). It runs on an IBM compatible PC and uses 3 input files, detailing the JTAG scan register bits (jtagscan.txt) and JTAG instruction opcodes (jtaginst.txt) specific to the processor used, and a Motorola standard s-record file (srec.abs) with the desired memory contents respectively. Based on these input files, the programmer downloads suitable JTAG sequences via the PC parallel port to manipulate and control the I/O of the processor on the target system. The aim is to program the processor’s JTAG scan register with defined states, before driving out the levels onto the processor pins and replicate the bus cycles (albeit slowly) which program system Flash or SRAM memory.

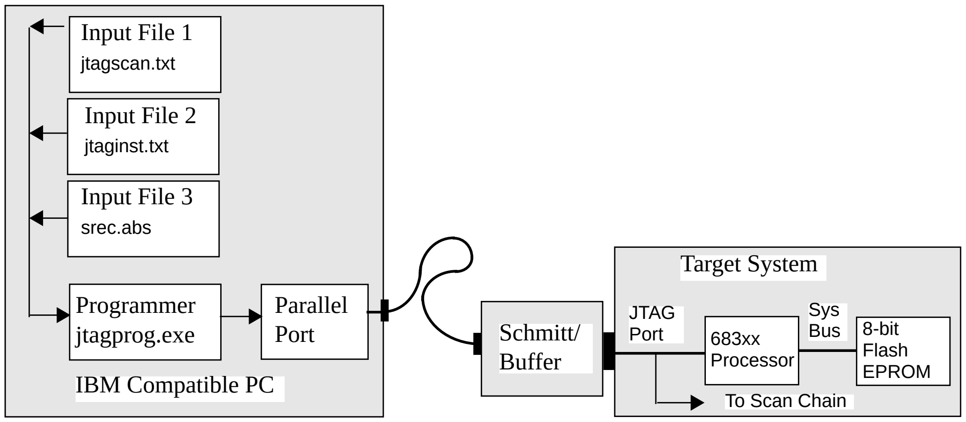

Figure 1. JTAG Flash Programmer Overview

Reproducing a bus cycle to write to Flash memory requires two full scans of the processor JTAG scan data register - one to start the bus cycle and one to terminate it. Figures 2 and 3 illustrates the concept for read and write cycles respectively.

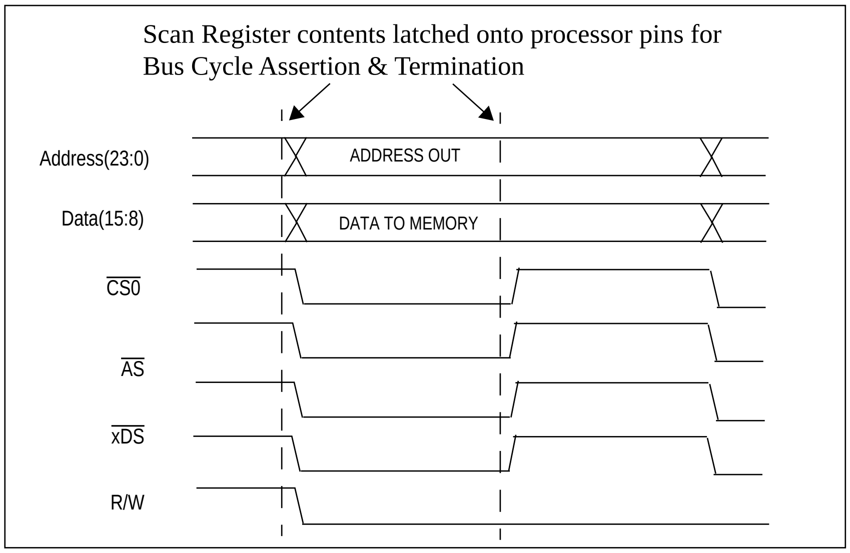

Figure 2. Memory Write Cycle Example

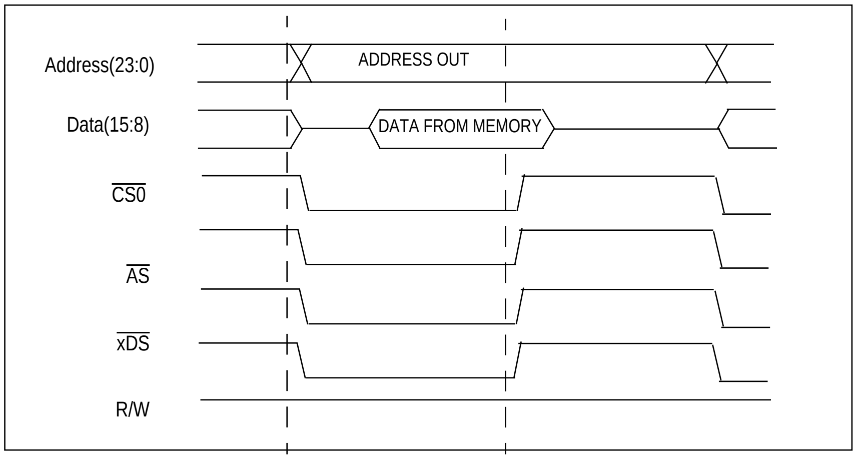

Figure 3. Memory Read Cycle Example

One key attribute of the JTAG Flash programmer is its flexibility. For example the jtagscan.txt input file gives the user complete freedom in the definition of signal states during read and write bus cycles to tailor towards specific system hardware requirements. In the memory write cycle of Figure 2, R/W is maintained low during the chip select negation, ensuring the data direction integrity is maintained until the memory write concludes. Also, if the memory interface does not use $\overline{AS}$ and $\overline{xDS}$ as part of the hardware decode for memory accesses, they need not be asserted during reads/writes. So the conclusion is that the flexibility means the bus cycles produced can deviate from the normal operation of the processor bus cycles. Accordingly, care must be taken not to damage the hardware. During reads for example, the data bus must be tri-stated to allow the memory to drive the bus and avoid contention.

The JTAG programmer was also designed for simplicity. Having set up the two JTAG scan input files (jtagscan.txt and jtaginst.txt) once for a particular processor system, simply re-compile the desired target code to form a new s-record file and use the programmer to download the new release of software into the target Flash EPROM. It is well suited to system software upgrades in the field.

BOUNDARY SCAN INSTRUCTIONS AND DATA

The JTAG operation itself assumes compliance with the IEEE 1149.1 Test Access Port and Boundary Scan Architecture standard. Refer to this document for a full description of the standard.

The Flash programmer uses only the mandatory instructions PRELOAD and EXTEST. The Flash programmer uses the PRELOAD instruction mechanism to select the boundary scan shift register in the data path and initialise the state of each of its latched parallel outputs. With the boundary scan shift register selected as the serial data path between TDI and TDO, the EXTEST instruction then drives the levels in the scan register onto the processor pins. Thereafter, each time the scan register data is updated (in Update_DR TAP controller state) new levels are driven onto the processor pins. Two such updates are made to the scan register (while EXTEST instruction remains active) to replicate the start and end of a write (or read) bus cycle to memory.

On the completion of the Flash programmer JTAG sequence the TAP controller state diagram is placed back into the Test-Logic Reset state, prior to releasing the reset control signal at the PC parallel port. The processor can then start up as expected during normal (non test) operation and execute the new Flash firmware.

INPUT FILES

JTAG Scan Register Bit Input File (JTAGSCAN.TXT)

This file defines the values shifted into the boundary scan register when the TAP controller is in the Shift_DR state, making the states driven onto the processor pins completely flexible. The file has 6 columns of information:

```
/*  1) Scan reg    2) Bit   3) write       4)write      5)read         6)read
/*  Cell name      Number   active state   end state    active state   end state
/*  ===========    ======   ============   =========    ============   =========
```

1.) The scan register cell name is in the main for the user's readability. The exceptional cases are that of address and data. They are searched out explicitly, and so must be named A0, A1, A2, etc., and D0, D1, D2...etc. A maximum limit of 32 address lines, and 32 data lines are expected by the programmer.

2.) The bit numbers should be contiguous (e.g. 0:115) and are used as a check that no bits have been omitted or added accidentally.

3.) The write active state is the level (1 or 0) of a particular scan register bit during the write. For a typical bus cycle, R/W, chip select x, upper data strobe could be active. (i.e. '0' level on pin), while lower data strobe may be inactive (i.e. '1' level on pin) because we have only an 8-bit memory on the upper half of the data bus.

4.) The write end state is the level (1 or 0) of a particular scan register bit at the end of a write. For a typical bus cycle, chip select x and upper data strobe would negate. (i.e. '1' level on pin).

5.) The read active state is the level (1 or 0) of a particular scan register bit during read cycles.

6.) The read end state is the level (1 or 0) of a particular scan register bit at the end of a read.

The only exceptional case for the read/write active and end state columns are address and data values that will vary on a cycle by cycle basis, and so cannot be given a specific level. They should be marked with an 'x'.The programmer detects occurrences of 'x' in the state values, and maps out the size of the address and data bus to be used based on this.

Example a) The address lines A(0:23) are marked as level 'x', data lines D(0:7) are marked as '0' and data lines D(8:15) are marked as 'x'. The Flash programmer interprets this as an 8-bit memory placed on the upper data bus, and so will vary address A(0:23) and D(8:15) as dictated by the memory map from the srecord input file (SREC.ABS). The s-record file data is read as byte wide. D(0:7) will always be driven as ‘0’.

Example b) The address lines A(0:23) are marked as level ‘x’ and data lines D(0:15) are marked as ‘x’. In this case, the Flash programmer recognises that the memory is 16 bits wide and so reads word wide data from the s-record input file to write out to data lines D(0:15).

The arrangement described offers the user maximum flexibility, where bus control bits can be individually controlled, and data byte, word or long word bus sizes can be selected. However, during the Flash programming process, the states driven in/out of processor pins may differ significantly from those expected during normal (non test) operation. For this reason, the programmer uses a RESET control signal from the parallel port to maintain the processor internals in a known state while the Flash programmer is manipulating the processor pins using the JTAG port. However it is the users responsibility to ensure the pin output levels driven - based on the JTAG scan register bit input file (JTAGSCAN.TXT) - do not damage target system. In particular, take heed that all processor scan register bits must be accounted for in this file. The scan data is input to the processor serially, so 1 bit too many, too few or with a wrong value in the chain can have serious repercussions for the system hardware. The file will need changed for be on a system-by-system basis.

JTAG Instruction Opcode Input File (JTAGINST.TXT)

The JTAGINST.TXT file defines the boundary scan instruction opcodes used - EXTEST and PRELOAD. These are the values shifted into the processors boundary scan instruction register when the TAP controller is in the Shift_IR state. Both the number of and value of instruction opcode bits defined are important, and the file will change on a processor-by-processor basis. The format expected is as shown:

```
EXTEST     =    0000     /* EXTEST opcode   = 0000  */
PRELOAD    =    0010     /* PRELOAD opcode   = 0010  */
```

S-Record Absolute Memory Image (SREC.ABS)

S-records are a Motorola defined format for downloading/outputting memory map images. Dump utilities, debuggers, cross assemblers and linkers for Motorola based processors can all generate s-record files.

The typical user scenario is to use a cross compiler/linker to generate an s-record image of a code section. The Flash programmer can then take this s-record file directly and download it as part of the boundary scan serial pattern to program the image into Flash. It can handle s-record files composed of S1, S2, or S3 style records. Also, based on the data bus width set in the jtagscan.txt input file (byte, word or long), the data information read from the s-record file will be byte, word, or long word wide respectively.

OPTIONAL OUTPUT FILE (DEBUG.TXT)

The -DEBUGON command line switch can be used to generate boundary scan serial and memory map debug information in a file (debug.txt). There are three debug formats used in the file: a descriptor outlining the current boundary scan operation in progress; memory address and data information that is being written (or read) within the serial scan register data stream; and the serial scan data stream itself detailing the individual nibbles of data being written to the 4 parallel port lines (TCK, TDI,TMS and reset control). The debug.txt file is not generated by default.

COMMAND LINE OPTIONS

The default settings for the programmer are to program a blank Flash EPROM. However, in addition the user has some command line options available:

`c:> jtagprog`	Program a blank Flash EPROM using the defacto pc parallel port address.

Options:

`-DEBUGON`	Create file "debug.txt" with serial and memory map debug information.
`-P1`	Alternative parallel port address P1 (LPT1 for colour monitors).
`-P2`	Alternative parallel port address P2 (LPT2 for colour monitors).
`-SRAM`	Program an SRAM (rather than a Flash EPROM).
`-ERASEF`	Erase full Flash EPROM Chip.
`-COMPARE`	Compare memory contents to s-record file (srec.abs).
`-DISPLAY`	Display a specified memory block on-screen.
`-DIVIDER xx`	Slow down the rate of download via the parallel port by divider value xx hex. The minimum divider 0 is the default, and the maximum divider is FFFF hex.

e.g.
```
c:> jtagprog -DEBUGON -P1 -COMPARE
```

Compare the current memory contents to an s-record file, outputting the JTAG scan data to alternate parallel port address P1 and at the same time creates a "debug.txt" output file logging the activity.

DOWNLOAD TIME AND PC HARDWARE REQUIREMENTS

During the mass production of embedded systems, the download time is a key concern, because this can equates directly to production costs. The JTAG Flash programmer was designed for simplicity rather than maximum download speed. It is therefore recommended to program a bootloader into system Flash, such that the system can use a faster download mechanism (UART, I2C, SPI etc.) to program the remainder of the Flash.

The download speed is largely dependent upon the PC hardware used and the number of scan bits and bus width of the processor. Based on the 25MHz '486 IBM Thinkpad PC hardware used, the JTAG programmer approach can download:

MC68306 16-bit Flash programming mode	137 Byte/sec
MC68306 16-bit SRAM programming mode	685 Byte/sec
MC68307 8-bit Flash programming mode	73 Byte/sec
MC68307 8-bit SRAM programming mode	365 Byte/sec

The download speed of the SRAM mode is 5 times faster than Flash mode. The SRAM mode completes only one write access to program the required byte (word/long) data into memory, while the Flash programming mode on the other hand requires 3 control byte writes to memory, the byte (word/long word) write itself, and then a read access to poll the success of the write.

In terms of PC hardware, the JTAG Flash programmer should not require large amounts of RAM on the PC to run, no matter what the size of the s-record file being downloaded. Today the 4MB RAM typical on engineering PCs will be more than sufficient. In terms of disk space, the disk space required is as follows:

jtagprog.exe	-	32kB
jtagscan.txt	-	~10kB Typical, depending upon number of scan register bits and comments etc.
jtaginst.txt	-	~2kB Typical, depending upon number of comments etc.
srec.abs	-	dependent on user code size
debug.txt		Optional output file.

When the -DEBUGON option is used, the resultant “debug.txt” output file is generally large. The exact size is dependent upon the size of the user code downloaded and the number of scan register bits of the processor and also the mode which the JTAG programmer is running. The -DEBUGON option will also slow the download speed for a particular code segment due to the large amount of information written out to file.

Use the following examples as a rule of thumb for the debug.txt file size.

MC68306 16-bit Flash programming mode	~260 x size of srec.abs
MC68306 16-bit SRAM programming mode	~52 x size of srec.abs
MC68307 8-bit Flash programming mode	~500 x size of srec.abs
MC68307 8-bit SRAM programming mode	~100 x size of srec.abs

Finally, beware the power management features available on most portable PCs when using the JTAG programmer. As data is sent to the PC parallel port without any handshake response, the power management operation may impede the data transfer to the parallel port resulting in lost data. The recommended solution is to use a desktop PC, or turn off the power management feature of the portable and maintain its power supply all the time.

CONNECTION TO TARGET

The connection between pc parallel port and the target system Flash EPROM is shown in Figure 4. It consists of a total of 5 connections: 4 PC parallel port outputs pins (pins 2:5), for the boundary scan TCK, TDI,TMS and reset control respectively and 1 pc parallel port input pin (pin 10) for TDO.

The IBM Thinkpad pc used had >1us rise and fall times at then parallel port outputs, so Schmitt triggers (74ACT74) were used to remove potential signal bounce to the JTAG port (TCK is usually most susceptible). I do not believe that the use of Schmitt trigger buffers is mandatory, but it was considered prudent to buffer the PC parallel port from any potentially damaging short circuits on the target system's JTAG port. 74ACT04 inverters are used to ensure the signals from the parallel port to the target have the correct sense.

The RESET control signal (R_CTL*) from the parallel port is used to drive the RESET/HALT open drain pins of the processor. This signal maintains the processor internals in a known state while the Flash programmer is manipulating the processor pins using the JTAG port. Prior to the JTAG sequences, the R-CTL* signal is driven high to put HOLD the part in RESET. At the end of the JTAG sequences, the R_CTL* is negated to allow the system to boot-up. Again this is gearing for the download of a bootloader program. Once the bootloader is programmed into Flash, the system boots up executing the bootloader code, ready to download subsequent code via a faster route such as M-Bus (I2C), or UART etc.

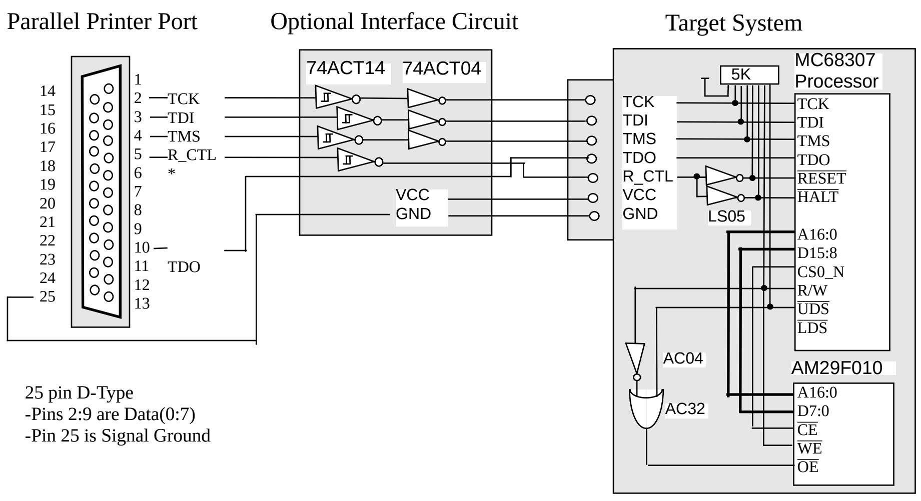

Figure 4. 8-Bit Flash Read/Write Circuit Hardware and Connectivity

The Flash programmer supports 8, 16 and 32-bit wide memory configurations.

MC68307 and 8-bit wide Flash EPROM

The interface between the MC68307 and 8-bit Flash memory Flash shown, cover the Flash programming and reading cases comprehensively. The configuration permits the JTAG port to program/read Flash, and the processor to program/read Flash if desired. It is the 16.67MHz MC68307 processor write to Flash that demands an EPROM access time of 120ns for the AM29F010 device. The timings are of course no problem when using the JTAG Flash programmer method of writing to Flash. In the MC68307 test case, the 8-bit data is driven on the upper half of the bus, but this may vary depending upon the processor used. The scan data defined for the write control bits in file jtagscan.txt is given in Listing 2.

MC68306 and 16-bit wide Flash EPROM

The MC68306 has a glueless interface to 16-bit Flash Memory, as shown in Figure 5. The MC68306 can read and write Flash via the JTAG or using the processor in normal mode. In either case the accesses can be either byte or word wide. Again, the Flash devices used are AM29F010 -120 for the 16.67MHz MC68306. The scan data defined for the write control bits in file jtagscan.txt is given in Listing 4.

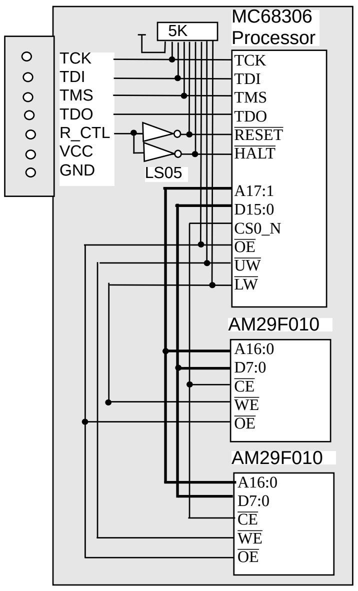

Figure 5. MC68306 Read/Write Interface to 16-Bit Flash Memory

SOFTWARE DESCRIPTION

The hierarchy of the jtagprog.c source code is illustrated in Figure 6. From left to right, the diagram represents each of the subroutines called directly from main() and thereafter successive layers of subroutine calls.

The jtagprog.c source code file itself is detailed in Listing 1, with comments documenting fully the structure. A synopsis of each routine is provided below for reference:

Routine	Scope
assert_reset()	Asserts reset control line (R_CTL*) prior to output of scan chain data.
binary_from_ulint()	Creates binary array of integer bits (e.g. A0..A32) from integer address.
clock_out_a_bit()	Outputs TCK low and then high phases for a TDI,TMS input combination.
do_write_bus_cycle()	Completes 2 scan chain updates for start and end of write bus cycles.
do_read_bus_cycle()	Completes 2 scan chain updates for start and end of write bus cycles.
extract_data_from_scan()	Used only during a read bus cycle to decipher read data from scan array.
fgetline()	reads one line of a specified file.
file_params_changed()	Changes cmd variable structure if command line options used.
get_jtag_opcodes()	Creates opcode variable structure based on jtaginst.txt file inputs.
get_scan_arrays()	Creates scan variable structure based on jtagscan.txt file inputs.
get_srec_line()	Copy next valid srecord line of file srec.abs.
get_srec_memdata()	Get the next byte/word/long data & address integers from an srecord line.
init_processor_bus()	Outputs PRELOAD and EXTEST instructions to JTAG TAP controller.
jtag_data_out()	Control output of a scan chain data array to TAP controller.
jtag_data_inout()	Control input & output of scan arrays (In on rising, Out on falling TCK).
jtag_instruct()	Control a TAP controller instruction execution.
jtag_reset()	Set TAP controller back to Test-Logic-Reset via 5 TCK periods with TMS=1.
main()	Overall control of programmer initialisation and modes.
negate_reset()	Negate reset control line (R_CTL*) as programmer completes.
output_to_port()	Outputs a byte to the parallel port with a programmable delay.
update_scan_memdata()	Update the read & write scan chain arrays with memory address and data.

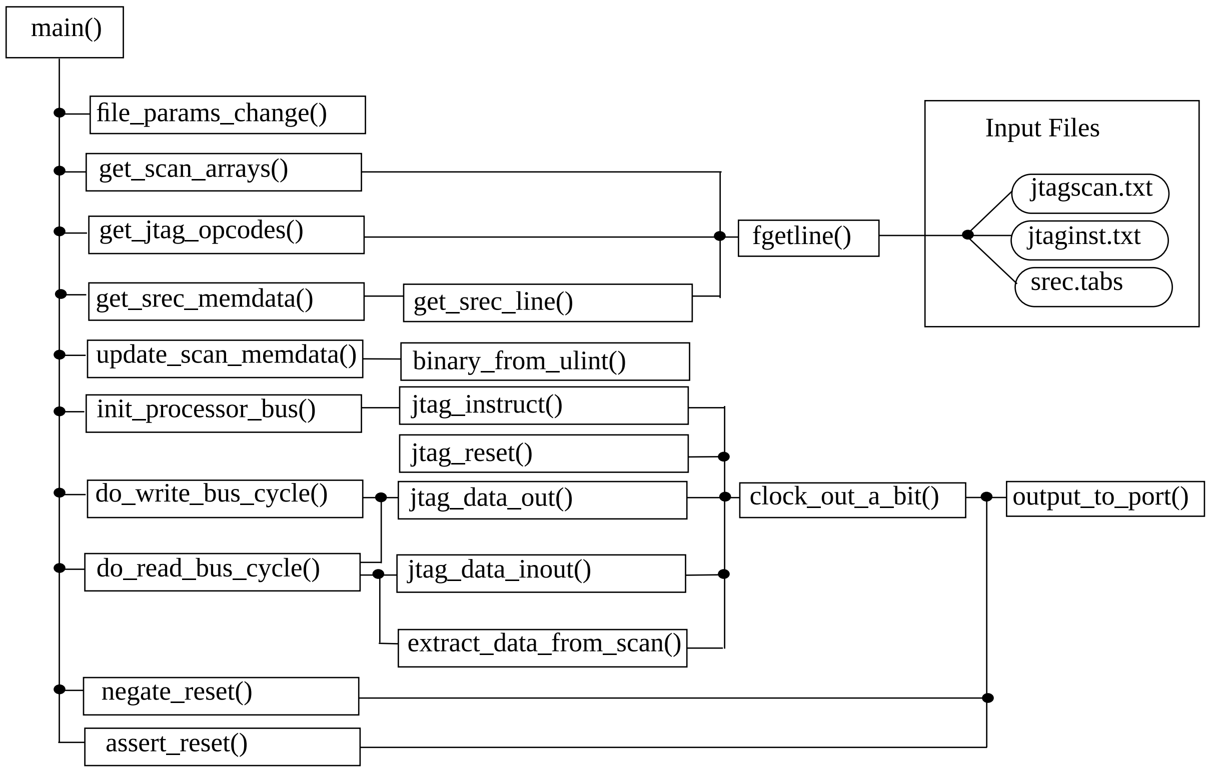

Figure 6. JTAGPROG.C Subroutine Hierarchy

FURTHER WORK

The Flash programmer is recommended as a system program bootloader. The bootload code programmed into Flash should be such that the system can use a faster download mechanism (UART, I2C, SPI etc.) to define the remainder of the Flash. This is one of several options available to improve the download speed.

The system code downloaded initially via the JTAG programmer could include the reset vectors, chip select initialisation and a slave MBus (also called I2C) driver. When the download of code is complete, the reset control is released to boot the system. The parallel port can then be used to implement a master MBus interface driving the remainder of the system code down for the processor to program into memory. With the capability of programming both Flash and SRAM via the JTAG programmer is down to user discretion as to whether the slave MBus code runs from Flash or SRAM. The PC parallel port hardware and connectivity required could be as shown in Figure 7 below:

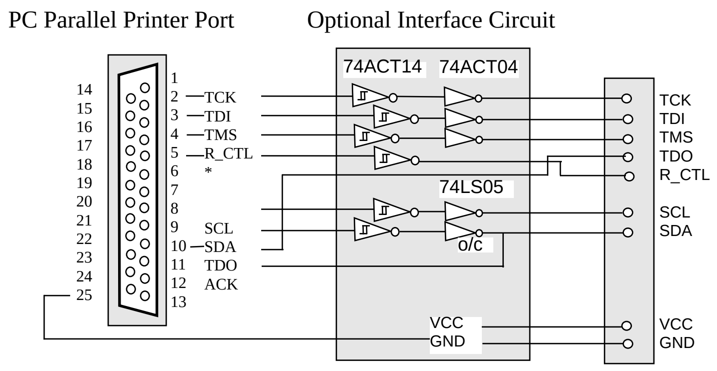

Figure 7. MBus and JTAG Parallel Port Connectivity

The MBus software driver would generate an interrupt on each received byte. Each time the MBus slave address is received the first 4 bytes could correspond to the load address and the subsequent bytes are loaded starting from that address. Assuming large contiguous memory areas are being downloaded, there is only a small address overhead (e.g. only 1 start address), so data rates approaching (100kBit per second /10 = 10kByte per second) are possible. Futhermore, the MBus implementation also provides an acknowledge (bit 9) handshake back from processor to parallel port confirming the safe reception of each data byte.

The benefit of the MBus implementation is that the parallel port completely defines the timing of via the SCL clock line. An SPI (serial peripheral interface) block has the same advantage. However, had a UART been selected it would need to transmit data out relative to a known baud rate input as shown in Figure 8 below. This requires the system hardware to possess not only a UART but also a timer reference clock output.

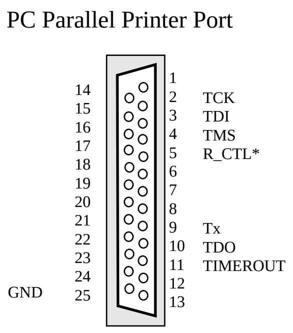

Figure 8. UART and JTAG Parallel Port Connectivity

As well as adopting these additional hardware blocks to improve the speed, JTAG programmer itself could have download times improved. Download speed could be increased dramatically by having all data to be downloaded ready in a file, such that it does not need manipulated, but simply written out to the parallel port. This approach was considered, but would consume too much in the way of disk space for the download data file to be a generic solution.

APPENDIX

File listings included for user reference are:

Listing 1 - JTAGPROG.C Source Code Listing
Listing 2 - MC68307 JTAGSCAN.TXT Input File Listing
Listing 3 - MC68307 JTAGINST.TXT Input File Listing
Listing 4 - MC68306 JTAGSCAN.TXT Input File Listing
Listing 5 - MC68306 JTAGINST.TXT Input File Listing

Listing 1 - JTAGPROG.C Source Code Listing

```
/* File    : JTAGPROG.C
   Purpose : Flash memory programming utility using JTAG
   Author  : James Gilbert
   Group   : High Performance Embedded Systems,
             Motorola,
             East Kilbride.
   Date    : 12th Sept 95
   Revision: 1.0
*/

/* Overview:
   To execute a single write access to memory, a minimum of 2 two full
   boundary scan register chain sequences are necessary to mimic the start
   & end of the bus access.The first drives the address, data, chip select,
   address and data strobes asserted for the active write portion. The
   second drives the same address and data, but with the chip select
   and strobes negated.

   The simplest way to do this is to alternate between 2 arrays of
   scan register values - active write values (swrite[]), and default
   values (swdef[]) for between writes. Each array is serialised
   bit-by-bit and output to target JTAG port via the PC parallel port

   Read accesses to memory follow an identical process to writes. The
   main difference is that read arrays are used for the read
   active start (sread[]) and the between reads (srdef[]) end of the
   bus cycle. The scan array data for reads must never allow the processor
   to drive data, only the address and control signals. The read data
   from the memory is latched during the Capture_DR state of the
   processors boundary scan state machine, as part of the bus cycle
   termination scanning.


   Parallel Port Connections:
   -------------------------------------------------------------
   | Parallel Port  : Parallel Port: Signal Name   : Direction |
   | Register Bit     Line Number                              |
   ------------------------------------------------------------|
   | base   (bit 0) : pin 2        : TDI           : Output    |
   | base   (bit 1) : pin 3        : TMS           : Output    |
   | base   (bit 2) : pin 4        : TCK           : Output    |
   | base   (bit 3) : pin 5        : RESET control : Output    |
   | base+1 (bit 6) : pin 10       : TDO           : Input     |
   -------------------------------------------------------------


   Step-by-Step through code:
   1)From scan register input file (SCAN_IFILE), the register settings
     required during and between writes (and reads) are determined.
     These are assigned to 2 write arrays - swrite[] and swdef[].
                       and 2 read  arrays - sread[]  and srdef[]
     Only address and data locations of the arrays will need to change
     for accesses to memory.

   2)Read the PRELOAD and EXTEST instruction opcodes from the
     INST_IFILE input file. This mechanism allows the number
```

of opcode bits to vary for different processors.

3)The bus and system state is initialised ready to run bus cycles
The PRELOAD instruction is used to select the boundary scan register and load the default bus state in the scan register before the EXTEST instruction drives the preloaded states out onto the processor pins.

4)Run Selected Bus Cycles

Modes: There are 5 primary Bus cycle modes controllable from the command line.

```
   0)          Default is FLASH Byte Program mode
   1)  -ERASEF  option for FLASH Chip Erase mode
   2)  -SRAM    option for SRAM Program mode
   3)  -COMPARE option for FLASH and SRAM compare mode
   4)  -DISPLAY option for displaying memory block contents
```

```
MODE0: The default mode is to program the flash memory with byte/word
       data. Before writing any user data to the flash device, the
       Flash must be put into its "Byte program mode".
       Thereafter the FLASH EPROM contents can be written.
       The successive adddress and data values to program into
       Flash are input by standard srecord file (SREC_IFILE).
       After each write of data to the Flash memory, the device
       must be read to poll whether he write is complete/successful.
       (WRITE and READ bus cyles)

MODE1: The ERASE FLASH mode writes a set of specified data
       to specific address to erase the previous FLASH chip contents.
       After the erase process, the Flash memory is polled to
       ensure success.
       (WRITE and READ bus cyles)

MODE2: In SRAM mode, there are no special writes for the FLASH,
       the memory data specified by the srec_ifile is sent immediately
       to memory. This is used for programming data into SRAM.
       (WRITE bus cyles only)

MODE3: Compare the memory contents to the srecord input file contents.
       The same process is used for both SRAM and FLASH memories.
       (READ bus cyles only)

MODE4: Display the contents of a memory block to the console.
       If the bus is 32-bit wide, memory is displayed long word wide.
       If the bus is 16-bit wide, memory is displayed word wide.
       If the bus is  8-bit wide, memory is displayed byte wide.
       (READ bus cyles only)
```

With memory map data available, and scan register arrays available for the 2 halves of a memory write access, the information is combined. This completes the array of binary scan bits for each half of the bus write (swrite[] & swdef[]). For reads, the processor data bus output buffers are disabled to allow the memory to drive the data. The two arrays used for each half of the read are (swrite[] & swdef[]).

The data is serialised and along with JTAG state control information is output on the pc parallel port (pins 2 to 5) to the JTAG port

```
     lines (see overview). With the EXTEST instruction still selected each
     update to the scan register contents is driven out onto the procesor
     pins.

   5)The JTAG control is placed back into the Test-Logic-Reset state
     and the RESET control to the processor negated, allowing the
     processor to start up and execute the newly loaded code in memory.
*/


#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

/* file names                                                          */
#define SCAN_IFILE         "jtagscan.txt"  /* scan reg detail input       */
#define INST_IFILE         "jtaginst.txt"  /* JTAG instr opcodes input    */
#define SREC_IFILE         "srec.abs"      /* srecord memory map input    */
#define DEBUG_OFILE        "debug.txt"     /* serial debug info out       */

/* file line lengths                                                   */
#define MAX_SREC_LEN       100             /* max length of srec line     */
#define MAX_SCAN_LEN       100             /* max length of scan line     */
#define MAX_INST_LEN       100             /* max length of instr line    */

/* file limitations                                                    */
#define FILENAME_MAX       3               /* max num of open files       */
#define MAX_AWIDTH         32              /* srec addr max 8 nibbles     */
#define MAX_DWIDTH         32              /* srec data max 8 nibbles     */
#define SREC_LINES_PER_DOT 2               /* set no of srec lines per '.'*/
#define MAX_SCAN_BITS      300             /* max num of scan reg bits    */
#define NAME_LEN           20              /* max scanbit/opcode name len */
#define OPCODE_LEN         10              /* max jtag instr opcode len   */

/* flash memory limitations                                            */
#define MAX_FLASH_POLL     1               /* max times Flash write polled*/
#define MAX_FLASH_EPOLL    1000            /* max times Flash erase polled*/
                                           /* erase in range 2 to 10 secs */
/* parallel port                                                       */
#define PARALLEL0          0x000003BC      /* parallel pt pntr0           */
#define PARALLEL1          0x00000378      /* parallel pt pntr1           */
#define PARALLEL2          0x00000278      /* parallel pt pntr2           */
#define SPEED_DIV          0x0             /* parallel pt speed           */
                                           /* slowest=FFFF                */

/* JTAG states                                                         */
#define Test_Logic_Reset   0x02            /* TMS (bit1) value to get     */
#define Run_Test_Idle      0x00            /* into the state named        */
#define Select_DR          0x02            /* Bit3 always=0,reset control */
#define Capture_DR         0x00
#define Shift_DR           0x00
#define Exit1_DR           0x02
#define Pause_DR           0x00
#define Exit2_DR           0x02
#define Update_DR          0x02
#define Select_IR          0x02
#define Capture_IR         0x00
#define Shift_IR           0x00
#define Exit1_IR           0x02
```

```
#define Pause_IR           0x00
#define Exit2_IR           0x02
#define Update_IR          0x12

/* RESET control                                                   */
#define Assert_Reset       0x00           /* Reset Control (bit3) low    */
#define Negate_Reset       0x00           /* Reset Control (Bit3) high   */

/* In future bring system out of reset by setting reset_control high      */
/* as follows. For now though maintain the reset low after completion     */
/* #define Negate_Reset    0x08                                    */


/***** VARIABLES (READ GLOBALLY) *******/
struct files_for_io                    /* file pntrs set in main()       */
{
  FILE *sfp;                           /* source file ptr            (R) */
  FILE *instfp;                        /* JTAG instr opcode file ptr (R) */
  FILE *scanfp;                        /* scan reg bits file ptr     (R) */
  FILE *dbugfp;                        /* serial out file pointer    (W) */
};
struct files_for_io fil;

struct cmd_line_params                 /* set in file_params_change()    */
{
  volatile char *parallel_port;        /* parallel pt pntr               */
  int           DEBUGON ;              /* Debug file creation OFF        */
  unsigned int  clk_divider;           /* div to speed/slow parallel data*/
  unsigned int  pmode;                 /* Program mode                   */
                                       /*    0=FLASH program mode        */
                                       /*    1=FLASH Erase mode          */
                                       /*    2=SRAm program mode         */
                                       /*    3=COMPARE memory to srecord */
                                       /*    4=DISPLAY memory contents   */
};
struct cmd_line_params cmd;            /* command line parameters        */


/***** VARIABLES (Parameter Passed) *******/
struct scan_variables                  /* vars set in get_scan_arrays() &*/
                                       /* updated in update_scan_memdata */
{
  char sname[MAX_SCAN_BITS][NAME_LEN];  /* scan bit name                 */
  int  snum[MAX_SCAN_BITS];            /*   "      number                */
  char swrite[MAX_SCAN_BITS];          /*   "      state during write    */
  char swdef[MAX_SCAN_BITS];           /*   "      state between writes  */
  char sread[MAX_SCAN_BITS];           /*   "      state during read     */
  char srdef[MAX_SCAN_BITS];           /*   "      state between reads   */

  unsigned int addr_bit[MAX_AWIDTH];    /* addr bit num within scan reg   */
  unsigned int data_bit[MAX_DWIDTH];    /* data bit num within scan reg   */

  int  num_of_scan_bits;               /* total scan bits num in scanfile*/
  int  lo_data_num;                    /* lowest data bus bit number     */
  int  hi_data_num;                    /* highest data bus bit number    */
  int  lo_add_num;                     /* lowest address bus bit number  */
  int  hi_add_num;                     /* highest address bus bit number */
};
```

```
struct jtag_opcodes
{
  char EXTEST[OPCODE_LEN];              /* EXTEST opcode pointer          */
  char PRELOAD[OPCODE_LEN];             /* PRELOAD opcode pointer         */
  int  len;                             /* number of bits per instr opcode*/
};

struct srec_variables                   /* srecord variables set in       */
                                        /* get_srec_memdata() routine     */
{
  volatile char line[MAX_SREC_LEN];     /* 1 line of srecord line array   */
  volatile int  len;                    /* length of srecord line         */
  volatile int  nextpos;                /* next char position within line */
  unsigned long address_int;            /* integer address of next byte   */
  unsigned long line_cnt;               /* number of srecord lines read   */
};


/***** FUNCTIONS*******/
int  file_params_change(int argc, char *argv[]);
int  get_scan_arrays( FILE *fp, struct scan_variables *s);
int  get_jtag_opcodes(FILE *fp, struct jtag_opcodes *op);
int  get_srec_memdata(FILE *fp, unsigned long *addr_int
                              , unsigned long *data_int
                              , int bussize
                              , struct srec_variables *sr);
int  get_srec_line(FILE *fp, unsigned long *address_int
                           , struct srec_variables *sr);
int  fgetline(FILE *fp, int limit,char *s);
void update_scan_memdata(unsigned long address_int
                        ,unsigned long data_int
                        ,struct scan_variables *s );
void binary_from_ulint(int len,unsigned long int ival,char *bin);
void do_write_bus_cycle(struct scan_variables *s);
unsigned long do_read_bus_cycle(struct scan_variables *s);
unsigned long extract_data_from_scan(unsigned char *scanreg
                                   ,struct scan_variables *s );
void ASSERT_RESET(void);
void NEGATE_RESET(void);
void init_processor_bus(struct scan_variables *s
                       ,struct jtag_opcodes *op);
void JTAG_RESET(void);
void JTAG_INSTRUCT(char *code, int len);
void JTAG_DATA_OUT(char *scan_reg_o,int reglen);
unsigned char *JTAG_DATA_INOUT(char *scan_reg_o,int reglen);
void clock_out_a_bit(char outchar);
void output_to_port(char outchar);
int  strcmp(char *,char *);



main(int argc, char *argv[])
{
  struct scan_variables scan,*s =&scan;   /* scan register variables pntr */
  struct jtag_opcodes opcode,*op=&opcode; /* jtag instr opcode struct pntr*/
  struct srec_variables srec,*sr=&srec;   /* srecord variables struct pntr*/

  int           i,j;                      /* loop count                   */
  int           err;                      /* temporary error code value   */
  int           err_store=0;              /* error code store=1 if error  */
  unsigned long datar_int;                /* data read from memory        */
  unsigned long data_int;                 /* 1 srecord data byte/word     */
  unsigned long addr_int;                 /* 1 srecord address            */
  unsigned long starta,enda;              /* start/end address for DISPLAY*/

  /* Initial bus state */
  unsigned long init_statea = 0;          /* Initial address=0            */
  unsigned long init_stated = 0;          /* Initial data=0               */

  /* FLASH byte program sequence  */
  unsigned long fproga[] = {0x5555,0x2AAA,0x5555};
  unsigned long fprogd[] = {0xAAAA,0x5555,0xA0A0};
  /* FLASH read/RESET sequence  */
  unsigned long freseta[]= {0x5555,0x2AAA,0x5555};
  unsigned long fresetd[]= {0xAAAA,0x5555,0xF0F0};
  /* FLASH Chip erase sequence  */
  unsigned long ferasea[]= {0x5555,0x2AAA,0x5555,0x5555,0x2AAA,0x5555};
  unsigned long ferased[]= {0xAAAA,0x5555,0x8080,0xAAAA,0x5555,0x1010};

  /* Initialise the Command Line Parameter Defaults                       */
  cmd.parallel_port = (char *)PARALLEL0;  /* Set parallel pt default addr */
  cmd.clk_divider   = SPEED_DIV;          /* Set parallel pt speed=slowest*/
  cmd.DEBUGON       = 0;                  /* Debug mode is off            */
  cmd.pmode         = 0;                  /* 0=FLASH program mode         */
                                          /* 1=FLASH Erase mode           */
                                          /* 2=SRAM program mode          */
                                          /* 3=COMPARE memory contents    */
                                          /* 4=DISPLAY memory block       */

  /* Verify if parameters passed from command line requiring              */
  /* changes to system setup. Debug file creation, parallel               */
  /* port address & download speed can be changed                         */
  if (argc > 1)
    err_store=file_params_change(argc,argv);
  else
    printf("Default parameters assumed!\n");

  if (err_store==0)     /* No Errors reading from input files             */
  {
   printf("\n********************************************\n");
   printf("JTAG Flash Programming Utility. (Rev 1.0)\n");
   printf("Designed by Motorola, EKB, Scotland.\n");
   printf("James Gilbert, September 1995.\n");
   printf("(c)Copyright Motorola 1995.\n");
   printf("********************************************\n\n");
   printf("Program start!\n");

   /* Open source files to ensure scan, jtag instr & scan files present   */
   fil.scanfp =fopen(SCAN_IFILE, "r");
   fil.instfp =fopen(INST_IFILE, "r");
   fil.sfp    =fopen(SREC_IFILE, "r");

   if((fil.scanfp!=0) && (fil.instfp!=0) && (fil.sfp!=0))
   {
/**********************************************************************/
    /* 1) Read scan register details, and create scan arrays.            */

    printf("Read scan register bit assignments(file: %s)\n",SCAN_IFILE);
    err=get_scan_arrays(fil.scanfp,s),
    err_store=err_store | err;
    fclose(fil.scanfp);

    /* Check if 8 16 or 32 bit memory                                      */
    /* eg.If 16 bit memory, shift address for FLASH sequences (no A0 line)*/
    if ((s->hi_data_num-s->lo_data_num+1) > 8)
    {
      j = ((s->hi_data_num-s->lo_data_num+1) / 16);
      for (i=0;i<3;i++)
      {
        fproga[i]   = fproga[i]    << j;
        freseta[i]  = freseta[i]   << j;
        ferasea[i]  = ferasea[i]   << j;
        ferasea[i+3]= ferasea[i+3] << j;
      }
    }

    /* if 32-bit wide bus mirror the 16-bit data value to be 32 bits      */
    if ((s->hi_data_num-s->lo_data_num+1) > 16)
      for (i=0;i<3;i++)
      {
        fprogd[i]   += fprogd[i]    << 16;
        fresetd[i]  += fresetd[i]   << 16;
        ferased[i]  += ferased[i]   << 16;
        ferased[i+3]+= ferased[i+3] << 16;
      }
/**************************************************************************/
    /* 2) Read JTAG instruction opcodes                                    */

    printf("Read TAP Controller instr opcodes (file: %s)\n",INST_IFILE);
    err=get_jtag_opcodes(fil.instfp,op);
    err_store=err_store | err;
    fclose(fil.instfp);

    fclose(fil.sfp);

    /* ALL FILES CURRENTLY CLOSED                                          */

/**************************************************************************/
    /* 3) Initialise bus state ready to run bus cycles                     */

    if (err_store==0)     /* No Errors reading from input files           */
    {
      if (cmd.DEBUGON)
      {
       printf("Store scan/map Debug Information  (file: %s)\n",DEBUG_OFILE);
       fil.dbugfp=fopen(DEBUG_OFILE, "w+"); /* serial data out file (wr)*/
       fprintf(fil.dbugfp,"--------------------------------------------\n");
       fprintf(fil.dbugfp,"JTAG MEMORY PROGRAMMER DEBUGON - JG, Sept95\n");
       fprintf(fil.dbugfp,"(c)Copyright Motorola 1995.\n");
       fprintf(fil.dbugfp,"--------------------------------------------\n");
       fprintf(fil.dbugfp,"REM:Remember JTAG TAP controller instructions\n");
       fprintf(fil.dbugfp,"    and data scans are LSB first.\n");
       fprintf(fil.dbugfp,"KEY:Each hex character shown represents a \n");
       fprintf(fil.dbugfp,"    data nibble sent to the pc parallel port.\n");
       fprintf(fil.dbugfp,"    The parallel port reg bits 0:3 written \n");
       fprintf(fil.dbugfp,"    correspond to port pins 2:5 as follows:\n");
       fprintf(fil.dbugfp,"    5=RESET, 4=TCK, 3=TMS, 2=TDI\n\n");
      }
      /* Processor reset control Asserted                                */
      if (cmd.DEBUGON)
        fprintf(fil.dbugfp,"\nAssert RESET\n");
      ASSERT_RESET();

      /* Initialise scan register arrays (read and write) with addr=data=0*/
      update_scan_memdata(init_statea,init_stated,s);

      /* Initialise processor bus                                        */
      init_processor_bus(s,op);

      sr->nextpos = sr->len;              /* force get of first srecord   */
      sr->line_cnt=0;                     /* line count for '.' print = 0 */

/**************************************************************************/
/* 5 MODES: 0) FLASH Program                                             */
/*          1) FLASH Chip Erase                                          */
/*          2) SRAM Program                                              */
/*          3) COMPARE memory contents                                   */
/*          4) DISPLAY memory block                                      */
/**************************************************************************/
      /* 4) Run Bus Write Cycles in appropriate mode                     */

      switch (cmd.pmode)
      {
       case 0:        /* If in default FLASH PROGRAM mode                */
                      /* Output: the Byte Program Flash Sequence         */
                      /*       :  and memory map data                    */
                      /*       : the Read/Reset Flash Sequence           */

        fil.sfp    =fopen(SREC_IFILE, "r");
        printf("Output S-records to parallel port (file: %s)\n",SREC_IFILE);

        /* While valid srecord data exists, get the                      */
        /* string containing next address & data (memdata)               */
        while (get_srec_memdata(fil.sfp, &addr_int, &data_int,
                               (s->hi_data_num-s->lo_data_num+1),sr) == 0)
        {
          /* output the Flash Byte Program sequence for every address     */
          for (i=0; i<3; i++)
          {
            if (cmd.DEBUGON)
              fprintf(fil.dbugfp,"\n\nFlash= %lX %lX"
                                 ,fproga[i],fprogd[i]);
            update_scan_memdata(fproga[i],fprogd[i],s);
            do_write_bus_cycle(s);
          }

          if (cmd.DEBUGON)
            fprintf(fil.dbugfp,"\n\nWr A/D= %lX %lX",addr_int,data_int);

          /* update the scan register arrays with a byte address and data */
          update_scan_memdata(addr_int,data_int,s);
          /* Do actual write bus cycle using 2 EXTEST scans of arrays     */
          do_write_bus_cycle(s);

          /* Confirm data programmed into Flash properly                  */
          i=0;
          while(i<MAX_FLASH_POLL)           /* poll Flash maximum 2 times */
          {
            datar_int=do_read_bus_cycle(s); /* read from Flash - 2 scans  */
            if ((~(datar_int ^ data_int)) & 0x8080)
              break;                        /* valid write case           */
            else
              i++;                          /* try again                  */
          }

/* Exclude this check for the 68307. No TDO feedback due to errata        */
/*        if (i>=MAX_FLASH_POLL)
          {
            printf ("\nERROR! Flash Timing limit exceeded.\n");
            printf ("Address/Data = %lX %lX\n",addr_int,data_int);
            break;
          }*/
        }
        fclose(fil.sfp);

        printf("\nOutput Flash Read/Reset sequence to parallel port\n");
        for (i=0; i<3; i++)
        {
          if (cmd.DEBUGON)
            fprintf(fil.dbugfp,"\n\nFlash= %lX %lX",freseta[i],fresetd[i]);
          update_scan_memdata(freseta[i],fresetd[i],s);
          do_write_bus_cycle(s);
        }
        break;

/***************************************************************************/
       case 1:        /* If in FLASH Chip Erase mode                      */
                      /* Output: the Read/Reset Flash Sequence            */

        printf("Output Flash Chip Erase sequence to parallel port\n");
        for (i=0; i<6; i++)
        {
          if (cmd.DEBUGON)
            fprintf(fil.dbugfp,"\n\nFlash= %lX %lX",ferasea[i],ferased[i]);
          update_scan_memdata(ferasea[i],ferased[i],s);
          do_write_bus_cycle(s);
        }

        /* Confirm Flash erased properly                                  */
        printf("Polling for Flash Chip Erase complete (takes 2->10 secs)\n");
        i=0;
        while(i<MAX_FLASH_EPOLL)            /* poll Flash erase max limit */
        {
          datar_int=do_read_bus_cycle(s);   /* read from Flash            */
          if (datar_int & 0x8080)
            break;                          /* valid erase case           */
          else
            i++;                            /* try again                  */
        }

        if (i>=MAX_FLASH_EPOLL)
        {
          printf ("ERROR! Flash Erase Unsuccessful.\n");
          break;
        }
        printf("Output Flash Read/Reset sequence to parallel port\n");
        for (i=0; i<3; i++)
        {
          if (cmd.DEBUGON)
            fprintf(fil.dbugfp,"\n\nFlash= %lX %lX",freseta[i],fresetd[i]);
          update_scan_memdata(freseta[i],fresetd[i],s);
          do_write_bus_cycle(s);
        }

        break;


/************************************************************************/
       case 2:         /* If in SRAM mode                                */
                       /* Output: the s-record file data                 */

        fil.sfp    =fopen(SREC_IFILE, "r");
        printf("Output S-records to parallel port (file: %s)\n",SREC_IFILE);

        /* While valid srecord data exists,                              */
        /* get the string containing next address & data                 */
        while (get_srec_memdata(fil.sfp, &addr_int, &data_int,
                               (s->hi_data_num-s->lo_data_num+1),sr) == 0)
        {
          if (cmd.DEBUGON)
            fprintf(fil.dbugfp,"\n\nWr A/D= %lX %lX",addr_int,data_int);

          /* update the scan register arrays with a byte address and data */
          update_scan_memdata(addr_int,data_int,s);
          /* Do actual bus cycle using 2 EXTEST scans based on array input*/
          do_write_bus_cycle(s);
        }
        fclose(fil.sfp);
        break;

/************************************************************************/
       case 3:         /* If in Compare mode                             */
                       /* Output: Verify memory=s-record file data       */

        fil.sfp    =fopen(SREC_IFILE, "r");
        printf("Compare memory contents to S-record (file: %s) \n"
               ,SREC_IFILE);

        /* While valid srecord data exists,                              */
        /* get the string containing next address & data                 */
        while (get_srec_memdata(fil.sfp, &addr_int, &data_int,
                               (s->hi_data_num-s->lo_data_num+1),sr) == 0)
        {
          if (cmd.DEBUGON)
            fprintf(fil.dbugfp,"\n\nWr A/D= %lX %lX",addr_int,data_int);

          /* update the scan register arrays with a byte address and data */
          update_scan_memdata(addr_int,data_int,s);

          /* Confirm data in Flash properly = srecord data               */
          if ((datar_int=do_read_bus_cycle(s)) != data_int)
          {
            printf("\nERROR! Compare data does not match.\n");
            printf("Address     =%lX\n", addr_int);
            printf("Actual Data =%lX, while S-record Data =%lX\n",
                                  datar_int,data_int);
            break;
          }
        }
        printf("\nMemory Contents and S-record data are identical!\n");
        fclose(fil.sfp);
        break;

/***************************************************************************/
       case 4:        /* If in Memory Display mode                       */
                      /* Output: Memory contents shown on terminal        */
        printf("Enter Memory block start & end address (e.g. 0 FFFF)\n");
        if (scanf("%lx %lx",&starta,&enda) != 2)
        {
          printf("ERROR! Incorrect start/end address format\n");
          break;
        }

        /* take start address to multiple of 16                           */
        if ((starta % 16) != 0)
          starta= starta-(starta%16);

        /* Display memory map on screen in byte, word or long format      */
        for (addr_int=starta; addr_int<=enda
                            ; addr_int+=(s->hi_data_num-s->lo_data_num+1)/8)
        {
          if (cmd.DEBUGON)
            fprintf(fil.dbugfp,"\n\nRd A= %lX ",addr_int);

          /* update the scan register arrays with a byte address and data */
          update_scan_memdata(addr_int,0,s);
          datar_int=do_read_bus_cycle(s);

          /* if word wide data, address will add 2, else byte wide        */
          if ((s->hi_data_num-s->lo_data_num+1) == 32)
            if ((addr_int % 16) == 0)
             printf("\n%8.8lX   %8.8lX",addr_int,datar_int);
            else
             printf(" %8.8lX",datar_int);
          else if ((s->hi_data_num-s->lo_data_num+1) ==16)
            if ((addr_int % 16) == 0)
             printf("\n%8.8lX   %4.4lX",addr_int,datar_int);
            else
             printf(" %4.4lX",datar_int);
          else       /* byte format */
            if ((addr_int % 16) == 0)
              printf("\n%8.8lX   %2.2lX",addr_int,datar_int);
            else
              printf(" %2.2lX",datar_int);
        }
        printf("\n");
        break;

      default: break;
      } /* end switch */
```

```
/*********************************************************************/
      /* 5)The JTAG control is placed back into the Test-Logic-Reset state*/
      /*   and the RESET control to the processor negated.               */

      if (cmd.DEBUGON)
        fprintf(fil.dbugfp,"\nJTAG RESET\n");
      JTAG_RESET();         /* JTAG sequence for Test-Logic-Reset State   */

      if (cmd.DEBUGON)
        fprintf(fil.dbugfp,"\nNegate RESET\n");
      NEGATE_RESET();       /* Processor reset control negated            */

      if (cmd.DEBUGON)
        fclose(fil.dbugfp);

      printf("\nProgram complete!\n\n");
     }
     else
      printf("ERROR! Program not Complete. Incorrect Input.\n");
   }
   else
   {
     printf("ERROR! input file not found.\n");
     printf("Check files: %s, %s, %s\n",SREC_IFILE,SCAN_IFILE,INST_IFILE);
   }
  }
  exit(0);
  return 0;
}

/* Routine: file_params_change()                                         */
/* Scope  : On receipt of additional command line arguments,             */
/*          change the cmd structure globals as appropriate              */
/* Inputs : argc   - number of arguments from command line               */
/*          argv[] - array of arguments                                  */
/* Outputs: cmd structure global variables updated                      */
/*          returnval - returns error code (0 for o.k., 1 for error      */
int file_params_change(int argc,char *argv[])
{
  unsigned char *dbug    = "-DEBUGON";    /* Debug info to file parameter */
  unsigned char *pport1  = "-P1";         /* Change parallel port addr 1  */
  unsigned char *pport2  = "-P2";         /* Change parallel port addr 2  */
  unsigned char *erase   = "-ERASEF";     /* FLASH memory erase only      */
  unsigned char *sramm   = "-SRAM";       /* Program SRAM instead of FLASH*/
  unsigned char *compare = "-COMPARE";    /* Compare memory contents mode */
  unsigned char *display = "-DISPLAY";    /* Display memory contents mode */
  unsigned char *divider = "-DIVIDER";    /* Paralel Port clock divder    */
  int  returnval=0;                       /* error code, 1 means error    */
  int i;                                  /* loop count                   */

  for (i=1; i<argc; i++)
  {
    if (strcmp(dbug,argv[i]) == 0)
    {
      printf("DEBUGON accepted!\n");
      printf("Debug Information sent to file - %s\n",DEBUG_OFILE);
      cmd.DEBUGON=1;
    }
    else if (strcmp(pport1,argv[i]) == 0)
    {
      printf("Parallel port address change accepted!\n");
      printf("New Address is %Xh (typically colour LPT1)\n",PARALLEL1);
      cmd.parallel_port = (char *)PARALLEL1;      /* Set parallel pt to 1 */
    }
    else if (strcmp(pport2,argv[i]) == 0)
    {
      printf("Parallel port address change accepted!\n");
      printf("New Address is %Xh (typically colour LPT2)\n",PARALLEL2);
      cmd.parallel_port = (char *)PARALLEL2;      /* Set parallel pt to 2 */
    }
    else if ((strcmp(erase,argv[i]) == 0) && (cmd.pmode == 0))
    {
      printf("FLASH EPROM erase only accepted!\n");
      cmd.pmode = 1;                              /* Set FLASH erase mode */
    }
    else if ((strcmp(sramm,argv[i]) == 0) && (cmd.pmode == 0))
    {
      printf("SRAM Mode change accepted!\n");
      cmd.pmode = 2;                              /* Set SRAM mode on     */
    }
    else if ((strcmp(compare,argv[i]) == 0) && (cmd.pmode == 0))
    {
      printf("Compare Mode change accepted!\n");
      cmd.pmode = 3;                              /* Set Compare mode on  */
    }
    else if ((strcmp(display,argv[i]) == 0) && (cmd.pmode == 0))
    {
      printf("Display Mode change accepted!\n");
      cmd.pmode = 4;                              /* Set Display mode on  */
    }
    else if (strcmp(divider,argv[i]) == 0)
    {
      if (sscanf(argv[i+1],"%x",&(cmd.clk_divider)) != 1)
        cmd.clk_divider   = SPEED_DIV;    /* Set parallel pt speed=slowest*/
      printf("Clock prescaler for parallel port change accepted!\n");
      printf("New divider is %Xh \n",cmd.clk_divider);
      i++;        /* inc i because "-DIVIDER" and hex val treated as pair */
    }
    else
    {
      printf("Invalid option: %s\n",argv[i]);
      printf("Valid options are:\n");
      printf("                  : Default mode is Program Flash\n");
      printf("   -DEBUGON       : Output debug Information to file\n");
      printf("   -P1            : Change parallel port addr to 1\n");
      printf("   -P2            : Change parallel port addr to 2\n");
      printf("   -ERASEF        : Erase flash to blank (only)\n");
      printf("   -SRAM          : Program SRAM instead of FLASH\n");
      printf("   -COMPARE       : Compare memory contents\n");
      printf("   -DISPLAY       : Display memory contents\n");
      printf("   -DIVIDER 0     : Slow data speed to parallel port\n");
      printf("                    Default Value = 0\n");
      printf("                    Maximum value = FFFF\n");
      returnval=1;
    }
  }
  return returnval;
}
```

```
/* Routine: get_scan_arrays()                                           */
/* Scope  : Take the scan register input file, and extract the read/write */
/*          active and read/write inactive arrays. These are used for     */
/*          start and end respectively of reads/write bus cycles when     */
/*          latched onto processor pins.For those bits marked 'X' (addr & */
/*          data) record the scan register bit no. to permit update with  */
/*          memory map information later.                                 */
/* Inputs : *fp - pointer to file                                         */
/* Outputs: scan structure global variables updated.                      */
/*          returnval - returns error code (0 for o.k., 1 for error       */
int get_scan_arrays(FILE *fp,struct scan_variables *s)
{
  int  returnval=0;            /* error code returned, 1 means error     */
  int  acount=0;               /* address bit count - present & correct? */
  int  dcount=0;               /* data bit count - present & correct?    */
  int  temp_ad_num;            /* address or data bit num - temporary    */
  int  j,i=0;                  /* simple character counters              */
  char sline[MAX_SCAN_LEN];    /* 1 line of scan register file input     */
  char temp_ad_array[3];       /* address or data bit num - temporary    */
  char name[NAME_LEN];         /* name of scan bit,temporary char array  */

  s->lo_add_num=MAX_AWIDTH;    /* Initialise lower address bus bit num   */
  s->hi_add_num=0;             /*     "      upper address bus bit num   */
  s->lo_data_num=MAX_DWIDTH;   /*     "      lower data bus bit num      */
  s->hi_data_num=0;            /*     "      upper data bus bit num      */

  /* Create scan register array, based on SCAN_IFILE input               */
  while(fgetline(fp,MAX_SCAN_LEN,sline)>0)
  {
    /* not a comment, so valid scan bit                                  */
    if ((sline[0] != '/') && (sline[0] != ' '))
    {
      /* read scan info - bit name, number, write & default states       */
      if (sscanf(sline,"%s %d %c %c %c %c"
                                  ,name
                                  ,&(s->snum[i])
                                  ,&(s->swrite[i])
                                  ,&(s->swdef[i])
                                  ,&(s->sread[i])
                                  ,&(s->srdef[i])) != 6)
      {
        printf("ERROR! Scan file field missing!\n");
        printf("On Line = %s\n", sline);
        returnval=1;
      }

      for (j=0; j<NAME_LEN; j++)
        s->sname[i][j]=name[j];
      i++;
    }
    s->num_of_scan_bits=i;
  }

  /* confirm all scan bits present                                       */
  if ((s->snum[i-1]-s->snum[0]) != i-1)
  {
      printf("ERROR! Scan bit data missing\n");
      returnval=1;
  }

  /* Search out the unknown 'x' address and data bits                     */
  /* Store array of address and data scan reg bit numbers                 */
  for (j=0; j<s->num_of_scan_bits; j++)
  {
    switch (s->swrite[j])
    {
      case '0':
      case '1': s->swrite[j] =s->swrite[j] - '0';
                s->swdef[j]  =s->swdef[j]  - '0';
                s->sread[j]  =s->sread[j]  - '0';
                s->srdef[j]  =s->srdef[j]  - '0';
                break;

      case 'x':
        temp_ad_array[0]=s->sname[j][1];
        temp_ad_array[1]=s->sname[j][2];
        temp_ad_array[2]='\0';
        switch(s->sname[j][0])
        {
          case 'A':
          case 'a':

             if (sscanf(temp_ad_array,"%d",&temp_ad_num) ==1)
             {
               /* store scan reg bit number for address                   */
               s->addr_bit[temp_ad_num]=s->snum[j];

               /* Get address bus width                                   */
               if (temp_ad_num < s->lo_add_num)
                 s->lo_add_num = temp_ad_num;
               if (temp_ad_num > s->hi_add_num)
                 s->hi_add_num = temp_ad_num;

               /* increment address bit count                             */
               acount++;
             }
             else
             {
               printf("ERROR! Only Axx address line names! (e.g. A23)\n");
               returnval=1;
             }
             break;


          case 'D':
          case 'd':

             if (sscanf(temp_ad_array,"%d",&temp_ad_num)==1)
             {
               /* store scan reg bit number for data                      */
               s->data_bit[temp_ad_num]=s->snum[j];

               /* Get data bus width                                      */
               if (temp_ad_num < s->lo_data_num)
                 s->lo_data_num = temp_ad_num;
               if (temp_ad_num > s->hi_data_num)
```

```
                s->hi_data_num = temp_ad_num;

              /* increment data bit count                               */
              dcount++;
            }
            else
            {
              printf("ERROR! Only Dxx data line names! (e.g. D16)\n");
              returnval=1;
            }
            break;


          default : printf("ERROR! Only addr/data x values!\n");
                    returnval=1;
                    break;
        }
        break;

      default :  printf("ERROR! Invalid scan bit assignment.\n");
                 returnval=1;
                 break;
    }
  }

  /* confirm all address bus lines defined in scan register               */
  if ((s->hi_add_num - s->lo_add_num) != acount-1)
  {
    printf("ERROR! Incorrect address bus bits scan assignments.\n");
    returnval=1;
  }

  /* confirm all data bus lines defined in scan register                  */
  if ((s->hi_data_num - s->lo_data_num) != dcount-1)
  {
    printf("ERROR! Incorrect data bus bits scan assignments.\n");
    returnval=1;
  }

  /* confirm data bus is 8, 16 or 32 bit wide                             */
  if ((s->hi_data_num - s->lo_data_num + 1) % 8 != 0)
  {
    printf("ERROR! Data bus must be 8, 16 or 32-bits wide.\n");
    returnval=1;
  }
  return returnval;
}


/* Routine: get_jtag_opcodes()                                            */
/* Scope  : Get the PRELOAD and EXTEST JTAG instruction opcodes for TAP   */
/*          controller from an input file. Combined, they can drive states*/
/*          onto processor pins                                           */
/* Inputs : *fp - pointer to file                                         */
/* Outputs: opcode structure global variables updated                     */
/*          returnval - returns error code (0 for o.k., 1 for error       */
int get_jtag_opcodes(FILE *fp, struct jtag_opcodes *op)
{
  int  returnval=0;               /* error code returned, 1 means error   */
  int  count=0;                   /* opcode count                         */
  int  i;                         /* loop count                           */
  char line[MAX_INST_LEN];        /* 1 line of JTAG Instr opcode file     */
  char name[NAME_LEN];            /* instr name char array                */
  char code[OPCODE_LEN];          /* opcode char array                    */
  char dummyeq;                   /* removes = symbol                     */

  static char *extest  = "EXTEST";           /* pointer to EXTEST string  */
  static char *preload = "PRELOAD";          /* pointer to PRELOAD string */

  while(fgetline(fp,MAX_INST_LEN,line)>0)
  {
    /* not a comment, so valid instr opcode                               */
    if ((line[0] != '/') && (line[0] != ' '))
    {
      /* read instr info - instr name, equals, and opcode                 */
      if (sscanf(line,"%s %c %s",name,&dummyeq,code) == 3)
      {
        /* if name is "EXTEST", get the opcode                            */
        if (strcmp(name,extest) == 0)
        {
          for (i=0; ((i< OPCODE_LEN) && (code[i]!='\0')); i++)
            op->EXTEST[i]=code[i]-'0';
          op->len=i;
          count++;
        }

        /* if name is "PRELOAD", get the opcode                           */
        else if (strcmp(name,preload) == 0)
        {
          for (i=0; (i< OPCODE_LEN) && (code[i]!='\0'); i++)
            op->PRELOAD[i]=code[i]-'0';
          op->len=i;
          count++;
        }
      }
      else
      {
        printf("ERROR! JTAG Instruction declaration format error.\n");
        printf("On Line = %s\n", line);
        returnval=1;
      }
    }
  }

  if (count<2)              /* ensure correct number of opcodes received  */
  {
    printf("ERROR! JTAG Instruction opcode missing.\n");
    returnval=1;
  }

  return returnval;
}


/* Routine: get_srec_memdata()                                            */
/* Scope  : On successive calls read next data byte and byte address from */
/*          the current s-record line. If current character position      */
/*          in srecord line is at end of line, read the next line.        */
/* Inputs : *fp       - pointer to file                                 */
/*          bussize   - Multiple of 4 (e.g. 8, 16 or 32) bit wide data bus*/
/* Outputs: *addr_int - byte address integer pointer                    */
/*          *data_int - data byte/word integer pointers.                */
/*          srec structure global variables updated                     */
/*          returnval - returns error code (0 for o.k., 1 for error     */
int get_srec_memdata(FILE *fp, unsigned long *addr_int
                             , unsigned long *data_int
                             , int bussize
                             , struct srec_variables *sr)
{
  int  returnval=0;                     /* error code returned, 1=error  */
  int  j, nibbles;                      /* character loop counts         */
  int  spaces;                          /* memory location count         */
  unsigned long new_address_int;        /* address integer               */
  char data[(MAX_DWIDTH/4)+1];          /* data string 4 chars & /0 term */

  nibbles = bussize / 4;                /* 2 chars for 8bit,4 for 16bit..*/
  spaces  = bussize / 8;                /* no of memory spaces per access*/

  /* First time round, get an srecord line and start address             */
  if (sr->nextpos >= sr->len-2)
    returnval=get_srec_line(fp,&sr->address_int,sr);
  new_address_int=sr->address_int;      /* set current = saved address   */

  /* get desired number of nibbles. May need a new srec line to do so     */
  if (returnval==0)
  {
    for (j=0;j<nibbles;)
    {
      /* if the current byte position is at the end of the current        */
      /* srecord line, read in the next srecord line.                     */
      if (sr->nextpos >= sr->len-2)
      {
        returnval=get_srec_line(fp,&new_address_int,sr);

        if ((returnval==0) && (new_address_int < sr->address_int+spaces))
        /* yes the next byte of data is first on next line                */
          data[j++]=sr->line[sr->nextpos++];
        else if (returnval==0)
        {
          for (;j<nibbles;j++)          /* pad out missing byte with FFh */
            data[j]='F';
          break;                        /* exit outer for loop           */
        }
        else
          break;
      }
      else
        data[j++]=sr->line[sr->nextpos++];
    }
    data[j]='\0';
  }
  /* Write address and data Integers before return                        */
  sscanf(data,"%lx",data_int);          /*  read hex & make integer data */
  *addr_int=sr->address_int;

  if (  (new_address_int != sr->address_int)
     && (new_address_int >= sr->address_int+spaces))
    sr->address_int = new_address_int;   /* change to new addr            */
  else
    sr->address_int += spaces;           /* incr addr to next byte/word   */

  return returnval;
}


/* Routine: get_srec_line()                                               */
/* Scope  : Get one valid data line of srecord file                       */
/* Inputs : *fp - file pointer                                            */
/* Outputs: *address_int -  pointer to address from start of s-record line*/
/*          srec structure global variables updated                       */
/*          returnval - returns error code (0 for o.k., 1 for error       */
int get_srec_line(FILE *fp, unsigned long *address_int
                          , struct srec_variables *sr)
{
  int  returnval=0;                      /* error code returned, 1=error  */
  int addr_char_num=0;                   /* address char no for S1,S2,S3  */
  int j, charno;                         /* character loop counts         */
  char address[(MAX_AWIDTH/4)+1];

  /* continue trying to get a valid srecord line until                    */
  /* end of file, OR valid (S1, S2, S3 record)                            */
  while ((addr_char_num==0)
     && ((sr->len=fgetline(fp,MAX_SREC_LEN,sr->line))>0))
  {
    /* verify an srecord line                                             */
    if((sr->line[0]=='s') | (sr->line[0]=='S'))
    {
      switch(sr->line[1])
      {
        case '1': addr_char_num=4;
                  break;
        case '2': addr_char_num=6;
                  break;
        case '3': addr_char_num=8;
                  break;

        case '0': case '4': case '5': case '6':
        case '7': case '8': case '9':
                  addr_char_num=0;
                  break;

        default:  addr_char_num=0;
                  printf("ERROR! Invalid s-record type.\n");
                  break;
      }
    }
  }

  if ((sr->len % 2) != 0)
  {
    printf("ERROR! Invalid s-record line character count.\n");
    returnval=1;           /* otherwise return an error/finish            */
  }

/************************************************************************/
  /*  If new & valid srecord line, extract the start address              */
  /* Verify an srecord type has valid memory data (i.e. s1, s2 or s3)     */
  if(addr_char_num>0 && returnval==0)
  {
    /* inc num of s-rec lines read and print . every SREC_LINES_PER_DOT   */
    sr->line_cnt++;
    if(sr->line_cnt % SREC_LINES_PER_DOT == 0)
      printf(".");

    /* Now get number of address chars based on S1,S2,S3 record type      */
    j=0;
    for(charno=4; charno<4+addr_char_num;)
    {
      address[j]=sr->line[charno];
      charno++;
      j++;
    }
    address[j]='\0';     /* terminate string                              */
    /* resulting address integer will be incremented after each           */
    /* data byte of the srecord line is read                              */

    sscanf(address,"%lx",address_int); /* read hex & make integer addr    */
    sr->nextpos=charno;  /* remember next char position in srecord line   */
  }
  else
  {
    returnval=1;         /* otherwise return an error/finish              */
  }
  return returnval;
}


/* Routine: fgetline()                                                    */
/* Scope  : Get one non-blank line from input file                        */
/* Inputs : *fp   - file pointer                                          */
/*          limit - max size of line                                      */
/* Outputs: *s    - pointer to start of character string read             */
/*          return- length of line                                        */
int fgetline(FILE *fp,int limit,char *s)
{
  int c, i;
  for (i=0; i<limit-1 && (c=fgetc(fp)) != EOF && c != '\n'; ++i)
    *s++ = c;
  *s = '\0';
  return i;
}


/* Routine: update_scan_memdata()                                         */
/* Scope  : For a scanline, update the swrite swdef sread & srder arrays  */
/*          with binary address and data defined by memory map            */
/* Inputs : address_int - integer address                                 */
/*          data_int    - integer data byte/word                          */
/* Outputs: scan structure global variables updated                       */
void update_scan_memdata(unsigned long address_int
                        ,unsigned long data_int
                        ,struct   scan_variables *s )
{
  int i,j;                     /* loop count                              */
  char A[MAX_AWIDTH];           /* binary address bits                    */
  char D[MAX_DWIDTH];           /* binary data bits                       */

  /* From integer data create array of data bits D0..D31                  */
  binary_from_ulint(MAX_DWIDTH,data_int,&D[0]);

  /* From integer address create array of address bits A0..A31            */
  binary_from_ulint(MAX_AWIDTH,address_int,&A[0]);

  /* Correlate data bits to the unknown 'x' scan reg bit locations        */
  j = s->lo_data_num;         /* j=0 if low byte,j=8 if upper byte        */
  for (i=0;i<=(s->hi_data_num-s->lo_data_num);)
  {
    s->swrite[s->data_bit[j]] =D[i];
    s->swdef[s->data_bit[j]]  =D[i];
    s->sread[s->data_bit[j]]  =D[i];
    s->srdef[s->data_bit[j++]]=D[i++];
  }

  /* Correlate address bits to the unknown 'x' scan reg bit locations     */
  for (i=s->lo_add_num;i<=s->hi_add_num;i++)
  {
    s->swrite[s->addr_bit[i]]=A[i];
    s->swdef[s->addr_bit[i]] =A[i];
    s->sread[s->addr_bit[i]] =A[i];
    s->srdef[s->addr_bit[i]] =A[i];
  }
}

/* Routine: binary_from_ulint()                                           */
/* Scope  : Take integer address/data and generate array of binary bits   */
/* Inputs : len  - size of integer, (8, 16 or 32)                         */
/*          ival - integer address data                                   */
/* Outputs: *bit - pointer to binary array                                */
void binary_from_ulint(int len,unsigned long int ival,char *bit)
{
  int i;
  unsigned long bitmask=0x0001;
  unsigned long intbit;
  for(i=0; i<len; i++)
  {
    intbit = ival & bitmask; /* mask off all bits except lsb              */
    *bit=intbit;             /* integer(1 or 0) -> char                   */
    ival=ival>>1;            /* shift right to map next bit to bit0 mask  */
    bit++;                   /* increment pointer                         */
  }
}


/* Routine: do_write_bus_cycle()                                          */
/* Scope  : Output write active & inactive arrays, to mimic start and end */
/*          of Write bus cycle. EXTEST instr drives scan reg states onto  */
/*          pins.                                                         */
/* Inputs : cmd.DEBUGON            - Debug facility On/Off indicator      */
/*          fil.dbug               - debug output file                    */
/*          s->swrite              - write active scan register array     */
/*          s->swdef               - between writes register array        */
/*          s->num_of_scan_bits    - number of scan register bits         */
/* Outputs: None                                                          */
```

```
void do_write_bus_cycle(struct scan_variables *s )
{
  /* Write out the active write scan array data                          */
  /* then execute jtag EXTEST command                                    */

  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nScan Reg Data Wr1\n");
  JTAG_DATA_OUT(&(s->swrite[0]),s->num_of_scan_bits);

/***********************************************************************/
  /* Write out the "between writes" scan array data                      */
  /* then execute jtag EXTEST command                                    */

  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nScan Reg Data Wr2\n");
  JTAG_DATA_OUT(&(s->swdef[0]),s->num_of_scan_bits);
}


/* Routine: do_read_bus_cycle()                                          */
/* Scope  : Output write active & inactive arrays, to mimic start and end */
/*          of Read bus cycle. EXTEST instr drives scan reg states onto   */
/*          pins. The data bus must be tri-state to allow memory to drive */
/*          during read.The scan reg sampled during read returned as pntr */
/* Inputs : cmd.DEBUGON            - Debug facility On/Off indicator      */
/*          fil.dbugfp             - debug output file                    */
/*          s->sread               - read active scan register array      */
/*          s->srdef               - between reads register array         */
/*          s->num_of_scan_bits    - number of scan register bits         */
/* Outputs: readval                - integer data read                    */
unsigned long do_read_bus_cycle(struct scan_variables *s)
{
  unsigned char *scan_reg_i;
  unsigned long readval;

  /* Write out the active read scan array data                           */
  /* then execute jtag EXTEST command                                    */

  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nScan Reg Data Rd1\n");
  JTAG_DATA_OUT(&(s->sread[0]),s->num_of_scan_bits);

/***********************************************************************/
  /* Write out the "between reads" scan array data                       */
  /* then execute jtag EXTEST command                                    */
  /* In the process read back the binary array from Capture_Dr TAP control*/

  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nScan Reg Data Rd2\n");
  scan_reg_i=JTAG_DATA_INOUT(&(s->srdef[0]),s->num_of_scan_bits);

  readval=extract_data_from_scan(scan_reg_i,s);
  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nRd D= %lX",readval);

  return readval;             /* return integer of data read from memory  */
}
```

```
/* Routine: extract_data_from_scan()                                     */
/* Scope  : For a scan register read, extract the data bus value         */
/*          Only scan structure global variables used                    */
/* Inputs : *scanreg             - pointer to scan register array        */
/*          s->databit           - array of address and data bit nums    */
/*          s->hi_data_num       - high data bus line number             */
/*          s->lo_data_num       - low data bus line number              */
/* Outputs: data_int             - integer data read                     */
unsigned long extract_data_from_scan(unsigned char *scanreg
                                   ,struct scan_variables *s )
{
  int i,j;                              /* loop count                    */
  unsigned long bitmask=0x0001;         /* only lsb so mask others       */
  unsigned long intbit=0;               /* int value of one data bit     */
  unsigned long data_int=0;             /* int data total                */
  char D[MAX_DWIDTH];                   /* binary data bits              */

  /* Correlate data bits to the unknown 'x' scan reg bit locations        */
  j = s->lo_data_num;         /* j=0 if low byte,j=8 if upper byte        */
  for (i=0;i<=(s->hi_data_num-s->lo_data_num);)
    D[i++]=scanreg[s->data_bit[j++]];

  /* now extract the integer value of combined bits                       */
  for(i=0; i<=(s->hi_data_num-s->lo_data_num); i++)
  {
    intbit = D[i] & bitmask;            /* mask off all bits except lsb  */
    intbit = intbit<<i;                 /* get power of 2 integer        */
    data_int += intbit;                 /* add bit value to total        */
  }
  return data_int;            /* return integer of data read from memory  */
}


/* Routine: ASSERT_RESET()                                               */
/* Scope  : Before a jtag sequence, assert a control signal to put       */
/*          system into a known state (typically connected to RESET_N    */
/*          or RSTINof processor 683xx processor)                        */
/* Inputs : None                                                         */
/* Outputs: None                                                         */
void ASSERT_RESET(void)
{
  output_to_port(Assert_Reset);               /* RST_CNTL=0,TMS=0,TDI=0  */
/*  delay in here if required */
}


/* Routine: NEGATE_RESET()                                               */
/* Scope  : Negate RESET control to processor to start executing code    */
/* Inputs : None                                                         */
/* Outputs: None                                                         */
void NEGATE_RESET(void)
{
  output_to_port(Negate_Reset);               /* RST_CNTL=1,TMS=0,TDI=0  */
}


/* Routine: init_processor_bus()                                                 */
/* Scope  : Initialise state of procesor bus by outputting               */
/*          PRELOAD and EXTEST instructions to JTAG TAP controller       */
/* Inputs : op->PRELOAD   - PRELOAD instrution opcode                      */
/*          op->EXTEST    - EXTEST instruction opcode                      */
/*          s->srdef[]    - scan chain array for read default pin states  */
/*          cmd.DEBUGON   - Debug facility On/Off indicator                */
/* Outputs: None                                                           */
void init_processor_bus(struct scan_variables *s
                       ,struct jtag_opcodes   *op)
{
  /* JTAG sequence for Test-Logic-Reset State                              */
  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nJTAG RESET\n");
  JTAG_RESET();

  /* Output jtag PRELOAD instruction to select scan register               */
  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nPRELOAD Instruction Active\n");
  JTAG_INSTRUCT(op->PRELOAD,op->len);

  /* Output Initial scan register arrays to bus (default read array)      */
  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nScan Reg Default Written\n");
  JTAG_DATA_OUT(&(s->srdef[0]),s->num_of_scan_bits);

  if (cmd.DEBUGON)
    fprintf(fil.dbugfp,"\nEXTEST Instruction Active\n");
  JTAG_INSTRUCT(op->EXTEST,op->len);
}


/* Routine: JTAG_RESET()                                                   */
/* Scope  : Assert 5 TCK inputs with TMS=1 to take the JTAG state machine */
/*          into the Test-Logic-Reset state.                               */
/* Inputs : None                                                           */
/* Outputs: None                                                           */
void JTAG_RESET(void)
{
  int i;

  for(i=0; i<5; i++)
    clock_out_a_bit(Test_Logic_Reset);         /* RST_CNTL=0,TMS=1,TDI=0  */
}


/* Routine: JTAG_INSTRUCT()                                                */
/* Scope  : Output serial sequence for JTAG TAP Controller instr execution*/
/* Inputs : *code - instruction opcode pointer                             */
/*          len   - instruction opcode length                              */
/* Outputs: None                                                           */
void JTAG_INSTRUCT(char *code,int len)
{
  int i;

  clock_out_a_bit(Run_Test_Idle);              /* RST_CNTL=0,TMS=0,TDI=x  */
  clock_out_a_bit(Select_DR);                  /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Select_IR);                  /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Capture_IR);                 /* RST_CNTL=0,TMS=0,TDI=x  */
  clock_out_a_bit(Shift_IR);                   /* RST_CNTL=0,TMS=0,TDI=x  */

  /* clock out the instruction opcode, lsb is first                        */
  for(i=len-1; i>0; i--)
    clock_out_a_bit(Shift_IR | *(code+i));     /* RST_CNTL=0,TMS=0,TDI=?  */
                                               /* decrement opcode pnter  */

  clock_out_a_bit(Exit1_IR | *code);           /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Update_IR);                  /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Run_Test_Idle);              /* RST_CNTL=0,TMS=0,TDI=x  */
}


/* Routine: JTAG_DATA_OUT()                                               */
/* Scope  : Sequence to write scan register using JTAG TAP Controller.    */
/* Inputs : *scan_reg_o - pointer to scan register array output           */
/*          reglen      - length of register                              */
/* Outputs: None                                                          */
void JTAG_DATA_OUT(char *scan_reg_o,int reglen)
{
  int i;

  clock_out_a_bit(Run_Test_Idle);              /* RST_CNTL=0,TMS=0,TDI=x  */
  clock_out_a_bit(Select_DR);                  /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Capture_DR);                 /* RST_CNTL=0,TMS=0,TDI=x  */
  clock_out_a_bit(Shift_DR);                   /* RST_CNTL=0,TMS=0,TDI=x  */

  for(i=0; i<reglen-1; i++)
    /* output 1 full TCK period, and new TDI state when clock goes low    */
    clock_out_a_bit(Shift_DR | *scan_reg_o++); /* RST_CNTL=0,TMS=0,TDI=?  */

  clock_out_a_bit(Exit1_DR | *scan_reg_o);     /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Update_DR);                  /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Run_Test_Idle);              /* RST_CNTL=0,TMS=0,TDI=x  */
}


/* Routine: JTAG_DATA_INOUT()                                             */
/* Scope  : Sequence to read and write scan register using JTAG TAP       */
/*          controller. Input current scan reg bit value on rising TCK    */
/*          before bit output on falling TCK moves to next scan position  */
/* Inputs : *scan_reg_o       - pointer to scan register array output     */
/*          reglen            - length of register                        */
/*          cmd.parallel_port - parallel_port bas location                */
/* Outputs: scan_reg_i        - pointer to scan register array input      */
unsigned char *JTAG_DATA_INOUT(char *scan_reg_o,int reglen)
{
  int i;
  unsigned char scan_reg_i[MAX_SCAN_BITS];

  clock_out_a_bit(Run_Test_Idle);              /* RST_CNTL=0,TMS=0,TDI=x  */
  clock_out_a_bit(Select_DR);                  /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Capture_DR);                 /* RST_CNTL=0,TMS=0,TDI=x  */
  clock_out_a_bit(Shift_DR);                   /* RST_CNTL=0,TMS=0,TDI=x  */

/* At this point we are in the Shift_Dr state, and the TCK is high        */
/* from now, we   clock out EXTEST data on falling TCK edges (lsb first)  */
/* & at same time clock in  SAMPLE data on rising  TCK edges (lsb first)  */

  for(i=0; i<reglen-1; i++)
  {
    /* output 1 full TCK period, and new TDI state when clock goes low    */
    clock_out_a_bit(Shift_DR | *scan_reg_o);   /* RST_CNTL=0,TMS=0,TDI=?  */
    scan_reg_o++;

    /* get TDO sample now as TCK high                                     */
    scan_reg_i[i]=inportb(cmd.parallel_port+1) & 0x40;
    /* if TDO input to parallel port is 1/0 then make array bit 1/0       */
    if (scan_reg_i[i]>0)
      scan_reg_i[i]=1;
  }

  clock_out_a_bit(Exit1_DR | *scan_reg_o);     /* RST_CNTL=0,TMS=1,TDI=x  */

  /* get last TDO sample now as TCK high                                  */
  scan_reg_i[i]=inportb(cmd.parallel_port+1) & 0x40;
  if (scan_reg_i[i]>0)
    scan_reg_i[i]=1;

  clock_out_a_bit(Update_DR);                  /* RST_CNTL=0,TMS=1,TDI=x  */
  clock_out_a_bit(Run_Test_Idle);              /* RST_CNTL=0,TMS=0,TDI=x  */

  return scan_reg_i;   /* return pointer to sampled scan reg input array  */
}


/* Routine: clock_out_a_bit()                                             */
/* Scope  : Execute one TCK period (low then high) on each call           */
/*          New TDI and TMS values after TCK goes low                     */
/* Inputs : outchar       - character to send to paralel port             */
/*          cmd.DEBUGON   - Debug facility On/Off indicator               */
/*          fil.dbugfp    - debug output file                             */
/* Outputs: None                                                          */
void clock_out_a_bit(char outchar)
{
    /* drive char onto parallel port, bit2 cleared to 0, so TCK low       */
    output_to_port(outchar & 0xFB);
    if (cmd.DEBUGON)
      fprintf(fil.dbugfp,"%X",outchar);
    /* drive char onto parallel port, bit2 set to 1, so TCK high          */
    output_to_port(outchar | 0x04);
}


/* Routine: output_to_port()                                              */
/* Scope  : Output a byte to parallel output port                         */
/* Inputs : outchar         - character to send to paralel port           */
/*          cmd.clk_divider - divider to slow output to paralel port      */
/* Outputs: None                                                          */
void output_to_port(char outchar)
{
  unsigned long int i;
                                       /* delay data to parallel port     */
  for(i=0;i<cmd.clk_divider;i++)       /* clk_divider from command line   */
      ;
  outportb(cmd.parallel_port,outchar); /* drive char onto parallel port   */
}
```

Listing 2 - MC68307 JTAGSCAN.TXT Input File Listing

```
/*************************************************************
/* File      : JTAGSCAN.TXT                                  *
/* Purpose   : 68307 boundary scan register bits for 8-bit   *
/*             Flash EPROM Programming.                      *
/* Author    : James Gilbert                                 *
/* Group     : High Performance Embedded Systems,            *
/*             Motorola,                                     *
/*             East Kilbride.                                *
/* Date      : 8th Sept 95                                   *
/* Revision  : 1.0                                           *
/*                                                           *
/*************************************************************
/* Disclaimer: If inappropriate settings are defined for the *
/*             boundary scan register in this file,damage to *
/*             the target system hardware could result.      *
/*                                                           *
/*************************************************************
/* Notes    a) 1 and 0 states refer to level in boundary     *
/*             scan register                                 *
/*          b) Address and data lines MUST be named          *
/*             specifically as the tool searches for them    *
/*             by name e.g. A0 A1, A2 ...                    *
/*             They have state x for both read and write     *
/*          c) Only address and data have state x            *
/*            --------------------------------------------  *
/*  ***     d) FOR READS, THE DATA BUS MUST NOT BE DRIVEN.   *
/*             It must be tristated using a control bit.     *
/*             (See bits 98 and 107 below as an example.)    *
/*            --------------------------------------------  *
/*                                                           *
/*************************************************************
/* Format   : Comment lines begin with /                     *
/*            No blank lines allowed until end of file       *
/*            Valid data must have one line for each scan bit*
/*            and 4 column fields as shown                   *
/*                                                           *
/*************************************************************
/*
/* All control cells (.CTL) are active high
/*
/*
/* Scan reg  Bit   write   default  read    read default
/* Cell name No.   state   state    state   state
/* ========= ====  ======  =======  =====   ============
ALE          0       1       1        1        1
RD           1       1       1        1        1
WR           2       1       1        1        1
BUS.CTL      3       1       1        1        1
AS           4       0       1        0        1
UDS          5       0       1        0        1
LDS          6       0       1        0        1
RW           7       0       1        1        1
DTACK_O      8       1       1        1        1
DTACK_I      9       1       1        1        1
RW.CTL       10      1       1        1        1
HALT_O       11      1       1        1        1
```

```
HALT_I       12      1       1        1        1
RESET_O      13      1       1        1        1
RESET_I      14      1       1        1        1
/*
/* Chip select 0 used for ROM
CS0          15      0       1        0        1
CS1          16      1       1        1        1
CS2          17      1       1        1        1
CS3          18      1       1        1        1
CLKOUT       19      1       1        1        1
BUSW         20      1       1        1        1
ADB.CTL      21      1       1        1        1
A0           22      x       x        x        x
A1           23      x       x        x        x
A2           24      x       x        x        x
A3           25      x       x        x        x
A4           26      x       x        x        x
A5           27      x       x        x        x
A6           28      x       x        x        x
A7           29      x       x        x        x
A8           30      x       x        x        x
A9           31      x       x        x        x
A10          32      x       x        x        x
A11          33      x       x        x        x
A12          34      x       x        x        x
A13          35      x       x        x        x
A14          36      x       x        x        x
A15          37      x       x        x        x
A16          38      x       x        x        x
A17          39      x       x        x        x
A18          40      x       x        x        x
A19          41      x       x        x        x
A20          42      x       x        x        x
A21          43      x       x        x        x
A22          44      x       x        x        x
AB.CTL       45      1       1        1        1
A23          46      x       x        x        x
IRQ7         47      1       1        1        1
PA7.CTL      48      0       0        0        0
BGACK        49      1       1        1        1
PA6.CTL      50      0       0        0        0
BG           51      1       1        1        1
PA5.CTL      52      0       0        0        0
BR           53      1       1        1        1
PA4.CTL      54      0       0        0        0
TOUT2        55      1       1        1        1
PA3.CTL      56      0       0        0        0
TOUT1        57      1       1        1        1
PA2.CTL      58      0       0        0        0
CS2D         59      1       1        1        1
PA1.CTL      60      0       0        0        0
CS2C         61      1       1        1        1
PA0.CTL      62      0       0        0        0
CS2B         63      1       1        1        1
PB15.CTL     64      0       0        0        0
INT8         65      1       1        1        1
PB14.CTL     66      0       0        0        0
INT7         67      1       1        1        1
PB13.CTL     68      0       0        0        0
```

```
INT6         69      1       1        1        1
PB12.CTL     70      0       0        0        0
INT5         71      1       1        1        1
PB11.CTL     72      0       0        0        0
INT4         73      1       1        1        1
PB10.CTL     74      0       0        0        0
INT3         75      1       1        1        1
PB9.CTL      76      0       0        0        0
INT2         77      1       1        1        1
PB8.CTL      78      0       0        0        0
INT1         79      1       1        1        1
PB7.CTL      80      0       0        0        0
TIN2         81      1       1        1        1
PB6.CTL      82      0       0        0        0
TIN1         83      1       1        1        1
PB5.CTL      84      0       0        0        0
CTS          85      1       1        1        1
PB4.CTL      86      0       0        0        0
RTS          87      1       1        1        1
PB3.CTL      88      0       0        0        0
RXD          89      1       1        1        1
PB2.CTL      90      0       0        0        0
TXD          91      1       1        1        1
PB1.PU       92      0       0        0        0
PB1.CTL      93      0       0        0        0
SDA          94      1       1        1        1
PB0.PU       95      0       0        0        0
PB0.CTL      96      0       0        0        0
SCL          97      1       1        1        1
DHI.CTL      98      1       1        0        0
D15          99      x       x        x        x
D14          100     x       x        x        x
D13          101     x       x        x        x
D12          102     x       x        x        x
D11          103     x       x        x        x
D10          104     x       x        x        x
D9           105     x       x        x        x
D8           106     x       x        x        x
DLO.CTL      107     1       1        0        0
/*
/* 8-bit data on upper byte
D7           108     0       0        0        0
D6           109     0       0        0        0
D5           110     0       0        0        0
D4           111     0       0        0        0
D3           112     0       0        0        0
D2           113     0       0        0        0
D1           114     0       0        0        0
D0           115     0       0        0        0
```

Listing 3 - MC68307 JTAGINST.TXT Input File. Listing

```
/*************************************************************
/* File      : JTAGINST.TXT                                  *
/* Purpose   : 68307 JTAG TAP Controller Instruction opcodes.*
/* Author    : James Gilbert                                 *
/* Group     : High Performance Embedded Systems,            *
/*             Motorola,                                     *
/*             East Kilbride.                                *
/* Date      : 8th Sept 95                                   *
/* Revision  : 1.0                                           *
/*                                                           *
/*************************************************************
/*                                                           *
/* Protocol  : instruction name in UPPERCASE                 *
/*           : EXTEST and PRELOAD are mandatory              *
/*           : Instruction opcode is binary wih no spaces    *
/*           : Only first 3 fields of non-comment line are   *
/*             valid                                         *
/*                                                           *
EXTEST     = 0000     /* EXTEST  opcode= 0000 */
PRELOAD    = 0010     /* PRELOAD opcode= 0010 */
```

Listing 4 - MC68306 JTAGSCAN.TXT Input File Listing

```
/**************************************************************
/* File      : JTAGSCAN.TXT                                  *
/* Purpose   : 68306 boundary scan register bits for 16-bit  *
/*             Flash EPROM Programming.                      *
/* Author    : James Gilbert                                 *
/* Group     : High Performance Embedded Systems,            *
/*             Motorola,                                     *
/*             East Kilbride.                                *
/* Date      : 8th Sept 95                                   *
/* Revision  : 1.0                                           *
/*                                                           *
/**************************************************************
/* Disclaimer: If inappropriate settings are defined for the *
/*             boundary scan register in this file,damage to *
/*             the target system hardware could result.      *
/*                                                           *
/**************************************************************
/* Notes    a) 1 and 0 states refer to level in boundary     *
/*             scan register                                 *
/*          b) Address and data lines MUST be named          *
/*             specifically as the tool searches for them    *
/*             by name e.g. A0 A1, A2 ...                    *
/*             They have state x for both read and write     *
/*          c) Only address and data have state x            *
/*            --------------------------------------------   *
/*  ***     d) FOR READS, THE DATA BUS MUST NOT BE DRIVEN.   *
/*             It must be tristated using a control bit.     *
/*             (See bits 98 and 107 below as an example.)    *
/*            --------------------------------------------   *
/*                                                           *
/**************************************************************
/* Format   : Comment lines begin with /                     *
/*            No blank lines allowed until end of file       *
/*            Valid data must have one line for each scan bit*
/*            and 4 column fields as shown                   *
/*                                                           *
/**************************************************************
/*
/* Active low cells are marked _N.
/*
/*
/*
/* Scan reg  Bit   write   default  read    read default
/* Cell name No.   state   state    state   state
/* ========= ====  ======  =======  =====   ============
OP1          0       1       1        1        1
OP0          1       1       1        1        1
IP1          2       1       1        1        1
IP0          3       1       1        1        1
TXDB         4       1       1        1        1
RXDB         5       1       1        1        1
TXDA         6       1       1        1        1
RXDA         7       1       1        1        1
OPOE3_N      8       1       1        1        1
OP3          9       1       1        1        1
IP2          10      1       1        1        1
```

X1	11	1	1	1	1
PPOE8_N	12	1	1	1	1
PA0	13	1	1	1	1
PPOE9_N	14	1	1	1	1
PA1	15	1	1	1	1
PPOE10_N	16	1	1	1	1
PA2	17	1	1	1	1
PPOE11_N	18	1	1	1	1
PA3	19	1	1	1	1
PPOE12_N	20	1	1	1	1
PA4	21	1	1	1	1
PPOE13_N	22	1	1	1	1
PA5	23	1	1	1	1
PPOE14_N	24	1	1	1	1
PA6	25	1	1	1	1
PPOE15_N	26	1	1	1	1
PA7	27	1	1	1	1
PPOE0_N	28	1	1	1	1
PB0	29	1	1	1	1
PPOE1_N	30	1	1	1	1
PB1	31	1	1	1	1
PPOE2_N	32	1	1	1	1
PB2	33	1	1	1	1
PPOE3_N	34	1	1	1	1
PB3	35	1	1	1	1
PPOE4_N	36	1	1	1	1
PB4	37	1	1	1	1
PPOE5_N	38	1	1	1	1
PB5	39	1	1	1	1
PPOE6_N	40	1	1	1	1
PB6	41	1	1	1	1
PPOE7_N	42	1	1	1	1
PB7	43	1	1	1	1
IACK1_N	44	1	1	1	1
IACK4_N	45	1	1	1	1
IACK7_N	46	1	1	1	1
IRQ1	47	1	1	1	1
IRQ4	48	1	1	1	1
IRQ7	49	1	1	1	1
D0	50	x	x	x	x
D1	51	x	x	x	x
D2	52	x	x	x	x
D3	53	x	x	x	x
D4	54	x	x	x	x
D5	55	x	x	x	x
D6	56	x	x	x	x
D7	57	x	x	x	x
DOE	58	1	1	0	0
D8	59	x	x	x	x
D9	60	x	x	x	x
D10	61	x	x	x	x
D11	62	x	x	x	x
D12	63	x	x	x	x
D13	64	x	x	x	x
D14	65	x	x	x	x
D15	66	x	x	x	x
HIZ_N	67	0	0	0	0
BERR_N	68	1	1	1	1
DTACK_N	69	1	1	1	1

```
FC0          70      1       1        1        1
FC1          71      1       1        1        1
FC2          72      1       1        1        1
RESET_N      73      0       0        0        0
HALT_N       74      0       0        0        0
CLKOUT       75      1       1        1        1
BR_N         76      1       1        1        1
BG_N         77      1       1        1        1
BGACK_N      78      1       1        1        1
AS_N         79      0       1        0        1
RW           80      0       1        1        1
UDS_N        81      0       1        0        1
LDS_N        82      0       1        0        1
UW_N         83      0       1        1        1
LW_N         84      0       1        1        1
OE_N         85      1       1        0        1
DRAMWOE_N    86      1       1        1        1
DRAMW_N      87      1       1        1        1
DRAMOE_N     88      1       1        1        1
RAS1_N       89      1       1        1        1
RAS0_N       90      1       1        1        1
CAS1_N       91      1       1        1        1
CAS0_N       92      1       1        1        1
/*
/* Chip select 0 used for EPROM
CS0_N        93      0       1        0        1
CS1_N        94      1       1        1        1
CS2_N        95      1       1        1        1
CS3_N        96      1       1        1        1
CPMOE        97      1       1        1        1
/*
/* Address Bus
A20          98      x       x        x        x
A21          99      x       x        x        x
CSOE_N       100     0       0        0        0
A22          101     x       x        x        x
A23          102     x       x        x        x
A1           103     x       x        x        x
A2           104     x       x        x        x
A3           105     x       x        x        x
A4           106     x       x        x        x
A5           107     x       x        x        x
A6           108     x       x        x        x
A7           109     x       x        x        x
A8           110     x       x        x        x
A9           111     x       x        x        x
A10          112     x       x        x        x
A11          113     x       x        x        x
A12          114     x       x        x        x
A13          115     x       x        x        x
A14          116     x       x        x        x
A15          117     x       x        x        x
AOE_N        118     0       0        0        0
A16          119     x       x        x        x
A17          120     x       x        x        x
A18          121     x       x        x        x
A19          122     x       x        x        x
AMODE        123     0       0        0        0
```

Listing 5 - MC68306 JTAGINST.TXT Input File Listing

```
/**************************************************************
/* File      : JTAGINST.TXT                                   *
/* Purpose   : 68306 JTAG TAP Controller Instruction opcodes *
/* Author    : James Gilbert                                  *
/* Group     : High Performance Embedded Systems,             *
/*             Motorola,                                      *
/*             East Kilbride.                                 *
/* Date      : 8th Sept 95                                    *
/* Revision  : 1.0                                            *
/*                                                            *
/**************************************************************
/*                                                            *
/* Protocol  : Instruction name in UPPERCASE                  *
/*           : EXTEST and PRELOAD are mandatory               *
/*           : Instruction opcode is binary with no spaces    *
/*           : Only first 3 fields of non-comment line are    *
/*             valid                                          *
/*                                                            *
EXTEST     = 000     /* EXTEST  opcode= 0000 */
PRELOAD    = 110     /* PRELOAD opcode= 0010 */
```

Literature Distribution Centers:
USA/EUROPE: Motorola Literature Distribution; P.O. Box 20912, Arizona 85036. 1-800-441-2447
JAPAN: Nippon Motorola Ltd.; 4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141 Japan.
HONG KONG: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Center, No. 2 Dai King Street, Tai Po Industrial Estate,