Flash Programming Guide

Chapter 1. Introduction to Flash Programming

Chapter 2. Design For On-board Programming

Chapter 3. Flash70 Digital Tests

Chapter 4. Data Sources for Flash Programming

Chapter 5. VCL Syntax for Flash OBP

Chapter 6. Generating Flash Digital Tests

Chapter 7. Validating Tests for Production

Chapter 8. Series and Parallel Programming

HEWLETT® PACKARD

NOTICE

This manual is provided “as is” and is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein, nor for direct, indirect, general, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Printed in U.S.A.

U.S. Government Restricted Rights

The Software and Documentation have been developed entirely at private expense. They are delivered and licensed as “commercial computer software” as defined in DFARS 252.227-7013 (Oct 1988), DFARS 252.211-7015 (May 1991) or DFARS 252.227-7014 (Jun 1995), as a “commercial item” as defined in FAR 2.101(a), or as “Restricted computer software” as defined in FAR 52.227-19 (Jun 1987) (or any equivalent agency regulation or contract clause), whichever is applicable. You have only those rights provided for such Software and Documentation by the applicable FAR or DFARS clause or the HP standard software agreement for the product involved.

Insulation Rating for Wires Connected to the System

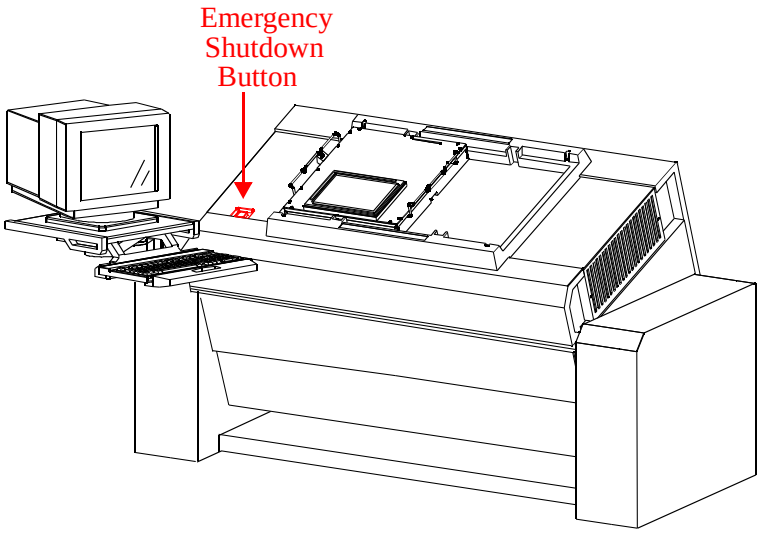
Use only external wiring with insulation rated for the maximum voltage (Vrms, Vpk or Vdc) and temperature to which the wire may be subjected in a fault condition.

Example: The system is connected to a source whose output is set at 50Vrms. The source could be set for as high as 300Vrms, intentionally or unintentionally. Therefore, the external wiring connected between this source and the system must be rated for 300Vrms.

HEWLETT PACKARD

NOTICE

Emergency Shutdown



The Emergency Shutdown switch is the large red button located at the lower left corner on the front of the testhead. It turns off all ac power to the testhead, and is equivalent to turning off the main circuit breaker on the rear of the pod. Press the Emergency Shutdown switch if you ever need to power down the testhead and its associated equipment in an emergency situation.

CAUTION: DO NOT use the Emergency Shutdown switch as a substitute for correct power-down procedures; i.e., executing the "testhead power off" command.

Frequent use of the Emergency Shutdown switch can cause premature failure of the main circuit breaker on the rear of the support bay.

To restore power after pressing the Emergency Shutdown switch, turn on the main circuit breaker on the rear of the pod.

HEWLETT® PACKARD

WARRANTY

1. HP warrants HP hardware, accessories and supplies against defects in materials and workmanship for the period of one year. If HP receives notice of such defects during the warranty period, HP will, at its option, either repair or replace products which prove to be defective. Replacement products may be either new or like-new.

2. HP warrants that HP software will not fail to execute its programming instructions, for the period of one year, due to defects in material or workmanship when properly installed and used. If HP receives notice of defects during the warranty period, HP will replace software media which does not execute its programming instructions due to such defects.

3. HP does not warrant that the operation of HP products will be uninterrupted or error free. If HP is unable, within a reasonable time, to repair or replace any product to a condition as warranted, customer will be entitled to a refund of the purchase price upon prompt return of the product to HP.

4. HP products may contain remanufactured parts equivalent to new in performance or may have been subject to incidental use.

5. The warranty period begins on the date of delivery or on the date of installation if installed by HP. If customer schedules or delays HP installation more than 30 days after delivery, warranty begins on the 31st day from delivery.

6. Warranty does not apply to defects resulting from (a) improper or inadequate maintenance or calibration, (b) software, interfacing, parts or supplies not supplied by HP, (c) unauthorized modification or misuse, (d) operation outside the published environmental specifications for the product, or (e) improper site preparation or maintenance.

7. TO THE EXTENT ALLOWED BY LOCAL LAW, THE ABOVE WARRANTIES ARE EXCLUSIVE AND NO OTHER WARRANTY OR CONDITION, WHETHER WRITTEN OR ORAL, IS EXPRESSED OR IMPLIED AND HP SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, SATISFACTORY QUALITY, AND FITNESS FOR A PARTICULAR PURPOSE.

8. HP will be liable for damage to tangible property per incident up to the greater of $300,000 or the actual amount paid for the product that is the subject of the claim, and for damages for bodily injury or death, to the extent that all such damages are determined by a court of competent jurisdiction to have been directly caused by a defective HP Product.

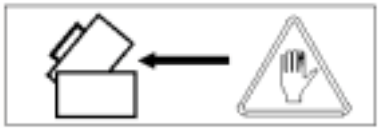
9. TO THE EXTENT ALLOWED BY LOCAL LAW, THE REMEDIES IN THIS WARRANTY STATEMENT ARE CUSTOMER'S SOLE AND EXCLUSIVE REMEDIES. EXCEPT AS INDICATED ABOVE, IN NO EVENT WILL HP OR ITS SUPPLIERS BE LIABLE FOR LOSS OF DATA OR FOR DIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL (INCLUDING LOST PROFIT OR DATA), OR OTHER DAMAGE WHETHER BASED IN CONTRACT, TORT, OR OTHERWISE.

FOR CONSUMER TRANSACTIONS IN AUSTRALIA AND NEW ZEALAND: THE WARRANTY TERMS CONTAINED IN THIS STATEMENT, EXCEPT TO THE EXTENT LAWFULLY PERMITTED, DO NOT EXCLUDE, RESTRICT OR MODIFY AND ARE IN ADDITION TO THE MANDATORY STATUTORY RIGHTS APPLICABLE TO THE SALE OF THIS PRODUCT TO YOU.

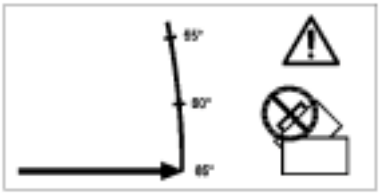
USER SAFETY SYMBOLS

These symbols are used on labels on various places on the testhead.

WARNING - Do not operate the testhead if you can see this symbol. It means that hazards exist because the safety shroud is not installed. These hazards include pinched fingers from pulling down a test fixture and electrical shock if HP Performance Port is installed.



WARNING - Keep your hands away from the indicated areas of the testhead to avoid pinched fingers when rotating the testhead.

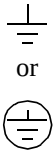


WARNING - Do not rotate the testhead past 65 degrees with a fixture installed, or the fixture could fall off the testhead, causing personal injury.

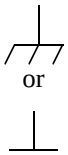
HEWLETT PACKARD

SAFETY SYMBOLS

Instruction symbol affixed to product. Indicates that the user must refer to the manual for specific WARNING and CAUTION information to avoid personal injury or damage to the product.



Indicates the field wiring terminal that must be connected to earth ground before operating the equipment - protects against electrical shock in case of fault.



Frame or chassis ground terminal -- typically connects to the equipment's metal frame.

Alternating current (ac).

Direct current (dc).

Indicates dangerous voltage.

WARNING Calls attention to a procedure, practice, or condition that could result in bodily injury or death.

CAUTION Calls attention to a procedure, practice, or condition that could cause damage to equipment or permanent loss of data.

HEWLETT PACKARD

WARNINGS

The following general safety precautions must be observed during all phases of operation, service, and repair of this product. Failure to comply with these precautions or with specific warnings elsewhere in this manual violates safety standards of design, manufacture, and intended use of this product. Hewlett-Packard Company assumes no liability for the Customer's failure to comply with these requirements.

Ground the Equipment: For Safety Class I equipment (equipment having a protective earth terminal), an uninterruptable safety earth ground must be provided from the main power source to the product input wiring terminals or supplied power cable.

DO NOT operate the product in an explosive atmosphere or in the presence of flammable gases or fumes.
For continued protection against fire, replace the line fuse(s) only with the fuse(s) of the same voltage and current rating and type. **DO NOT** use repaired fuses or short-circuited fuse holders.

Keep away from live circuits: Operating personnel must not remove equipment covers or shields. Procedures involving the removal of covers or shields are for use by service-trained personnel only. Under certain conditions, dangerous voltages may exist even with the equipment switched off. To avoid dangerous electrical shock, **DO NOT** perform procedures involving cover or shield removal unless you are qualified to do so.

DO NOT operate damaged equipment: Whenever it is possible that the safety protection features built into this product have been impaired, either through physical damage, excessive moisture, or any other reason, REMOVE POWER and do not use the product until safe operation can be verified by service-trained personnel. If necessary, return the product to Hewlett-Packard Sales and Service Office for service and repair to ensure that safety features are maintained.

Do not service or adjust alone: Do not attempt internal service or adjustment unless another person, capable of rendering first aid and resuscitation, is present.

Do not substitute parts or modify equipment: Because of the danger of introducing additional hazards, do not install substitute parts or perform any unauthorized modification to the product. Return the product to a Hewlett-Packard Sales and Service Office for service and repair to ensure that safety features are maintained.

Chapter 1
Introduction to Flash Programming

Rev. A

The information in this chapter provides an overview of flash RAM programming concepts. Read this chapter if you are unfamiliar with using automatic test equipment to program flash devices.

This chapter describes:

1.1 Flash RAM and On-board Programming

Flash RAM is a high-density, read-write, non-volatile memory source used in many electronics applications, such as digital cameras, modems, automotive engines, personal computers, and cellular phones. Unlike ROM, Flash memory is non-volatile, meaning data bits are retained even after removing the power supply. Additionally, flash memory is electrically erasable and re-writable in-system.

The utilization of flash memory devices by electronics manufacturers is on the rise. By the year 2000, the market for flash RAM is estimated to grow to $10 billion. For many manufacturers, flash RAM is becoming a standard element of circuit board design. Attributes such as in-system programmability and non-volatility make flash memory products a flexible, low cost, reliable memory solution for an increasingly diverse range of electronic products.

The value of using flash RAM over other programmable logic devices is that data can be electrically erased and re-programmed in-system. When choosing a manufacturing process for writing data to flash memory devices, project management teams should consider flash related manufacturing issues that impact product profitability. Inventory control procedures, the availability of third party vendors who can meet just-in-time goals, and the cost of acquiring special tools if Small Outline Packages (**SOP**s) are in the board design are important considerations. These issues add to the complexity of designing boards that include flash technology.

With the miniaturization of flash devices, traditional flash programming methods have become slow and expensive to implement. Manufacturers that program flash devices on PROM programmers often encounter damage rates as high as two percent because of the extra handling steps required to install the smaller, more fragile devices. Automatic test equipment (**ATE**) can be effectively utilized for flash on-board programming (**OBP**) to reduce costs and improve test development speed.

HP 3070 systems with Flash70 software can used for on-board programming of flash memory devices. This guide contains information about the procedures, tasks, and syntax required to perform flash programming with HP 3070 test systems. If flash programming is a new technology for you, then you should read this chapter, if you are familiar with flash technology, read about design for on-board programming requirements in **Chapter 2, "Design For On-board Programming."**

1.2 Flash Programming Concepts

You should be aware of the following concepts for programming flash memory devices:

- Flash memory is non-volatile.
- Flash memory is electrically erasable and writable.
- Non-blank flash devices must be erased prior to programming.
- Newer flash devices contain automated program verification procedures which reduce manual programming time and effort.
- Standardized libraries and algorithms can be used for programming flash devices.

Flash memory is a non-volatile, electrically erasable and writable memory originally developed by Intel® Corporation. Automatic programming modes in newer flash devices mean that flash algorithms are simpler and faster to implement. Like EPROM, flash memory devices must be erased before programming. HP Flash70 software manages flash programming activities at in circuit test, such as device erasure, verification and programming.

Programing methods for flash RAM vary based on part specifications and manufacturing requirements. **CFI,** the Common Flash Interface specification, enables the use of software algorithms for entire families of devices. CFI allows standardized software drivers to identify and use a variety of flash products because device identification data is embedded into the chip. Device identification data defines memory size, byte/word configuration, block configuration, and the voltages and timing information necessary for programming the device.

A variety of algorithms for programming flash devices exist. Intel® and Advanced Micro Devices (AMD®) have developed the algorithms most commonly used for flash RAM programming. Other manufacturers generally follow these industry leaders.

Automatic on-chip verification methods simplify the flash programming process and result in more reliable programming. Newer flash devices have internal automatic program verification processes. These devices manage data verification procedures by

automatically verifying the threshold levels of the data stored in the data cells. This increases testing efficiency because programmers no longer need to create separate threshold algorithms for every version of a flash device.

1.3 Flash Programming Tasks

Whether you are programming flash on-board with **ATE** or by some other method, the flash device programming process remains the same. In most test production environments, there are three steps:

1. **Device Identification**

   Identify the flash device by the manufacturer identification number.

   The flash programming mechanism reads the manufacturer identification number from the device to ensure the correct device is ready for programming.

2. **Device Erasuref**

   Verify that the device is blank. If not, perform device erasure.

   Flash devices must be blank before programming. New parts are shipped blank, but previously programmed parts must be erased prior to programming. Erasing a part changes all memory spaces on the flash device to a one state "1". This is necessary because the "program" command cannot program zero states to one**.**

NOTE

Ensuring that flash devices are blank before programming is one of the most overlooked steps for test developers.

3. **Device Programming**

   Program the flash device with valid information stored in a hex data record file.

   Flash devices are typically programmed with hex data records that follow either the Intel® Hex or Motorola® S-Record formats. Other formats exist, but they are not considered industry standards.

1.4 Embedded Programming Algorithms for Flash Devices

When 12.0 V flash memories were first introduced, cumbersome programming sequences were required to program and erase flash devices. To simplify the process, system software designers embedded flash programming algorithms onto newer flash devices. Embedded programming algorithms initialize, write, and read programming sequences automatically. In the HP 3070 environment, embedded algorithms are activated by the appropriate sequence of Vector Control Language (**VCL)** execute statements.

The two leading manufacturers of flash devices, Intel® and AMD® , use similar embedded programming algorithms. Flow charts that describe the write sequence algorithms follow. The procedures required to program a single byte of memory are indicated by the blocks in the diagram.

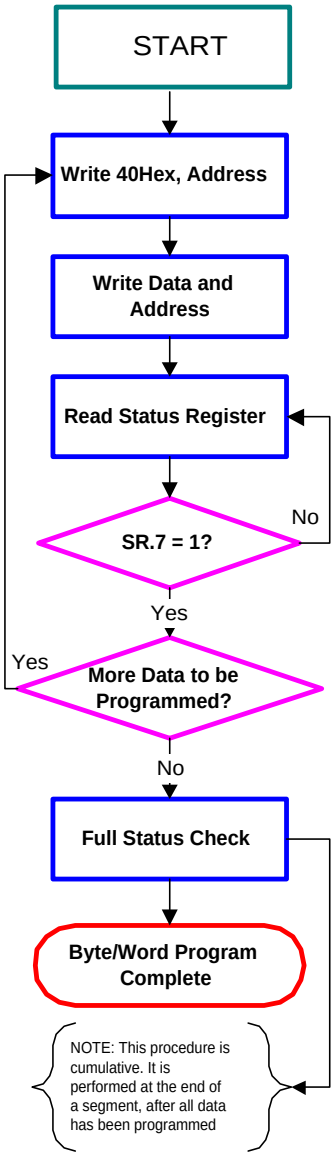
1.4.1 Intel® Algorithms

Intel® has developed algorithms designed to work with Intel® flash device architecture. Intel® flash devices contain a Command User Interface (CUI), which serves as the interface between the microprocessor and the internal operation of a flash device. Command sequences written to the CUI initiate embedded algorithms on flash devices. Valid command sequences cause an on-chip Write State Machine (WSM) to execute the algorithms and timing required to perform operations such as *Block Erase* and *Byte / Word Program*. The Write State Machine on the flash device manages block erase, program and lock-bit configuration functions.

1.4.2 Intel® Automated Byte/Word Programming Algorithm

Flash device programming can be executed with the Intel® *Byte/ Word Program* command. This involves writing two command sequences to the CUI: the Program Setup command (40H) and the address and data to be programmed. The Write State Machine then programs the data at specified address locations and verifies that programming was successful. When programmed, specified bits in an address location are changed to "0". The following flowchart illustrates the sequence of events that occur when the Byte/Word Program algorithm is executed.

1.4.2.1 Intel® Byte/Word Programming Flowchart



1. **Write 40H, Address**: *40H* is a write command for byte/word program setup. This command instructs the CUI to initialize the programming algorithm.

2. **Write Address and Data:** A second write command specifies the address and data to be written. The Write State Machine then controls program and program verify operations. Data is written to the cell and validated.

3. **Read Status Register:** The status register is read by writing the *Read Status Register* command. The status register determines when the program operation is completed successfully. After writing this command to the CUI, all subsequent read operations output data from the status register until another command is written to the CUI.

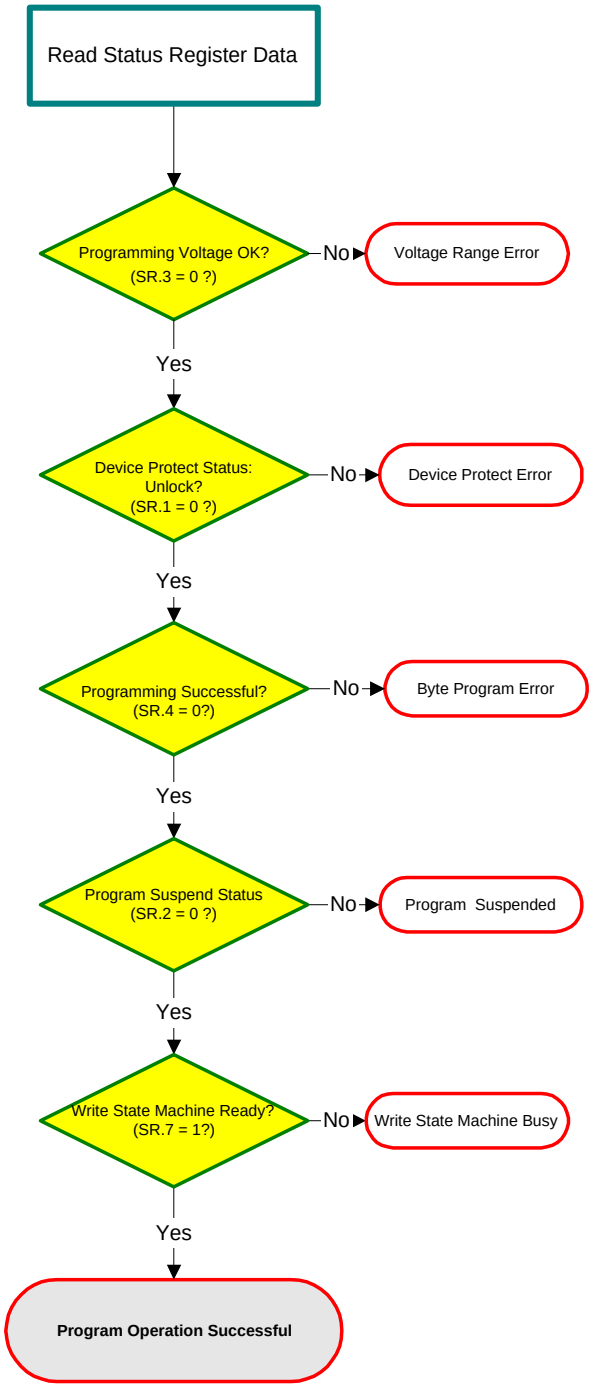
   The contents of the status register are latched on the falling edge of the **OE #** or the first edge of **CE #,** whichever occurs last in the read cycle.

4. **Verify Program Completion:** The status register is used to check the Write State Machine Status pin ("SR.7") for the following conditions:

   - If SR.7 = "0", the flash device is busy.
   - If SR.7 = "1" , the flash device program operation is complete and ready to receive more data.

5. **Repeat steps 1 through 4**, if there is more data.

6. **Perform Full Status Check:** A Full Status Check is performed after completing the device program operation.



IIntel® Full Status Check Algorithm

In the HP 3070 environment, a full status check reads the following status register pins to validate the programming algorithm:

SR.3 = Programming Voltage Status

•If SR.3 = "1", low programming voltage is detected and the program operation aborts.

•If the SR.3 bit equals "0", the programming operation is successful.

SR.1 = Device Protect Status

•If SR.1 ="1", a master lock-bit, block lock-bit and/or RP# lock is detected and the programming operation is aborted.

•If SR.1 = "0", this represents an unlocked bit.

SR.4 = Program and Set Lock-Bit Status

•If SR.4 = "1", an error in the Byte/Word programming occurred.

•If SR.4 = "0", the programming operation completed successfully:

SR.2 = Program Suspend Status

•If SR.2 = "1", the programming operation is suspended.

•If SR.2 = "0", the programming operation continues:

SR.7 = Write State Machine Status

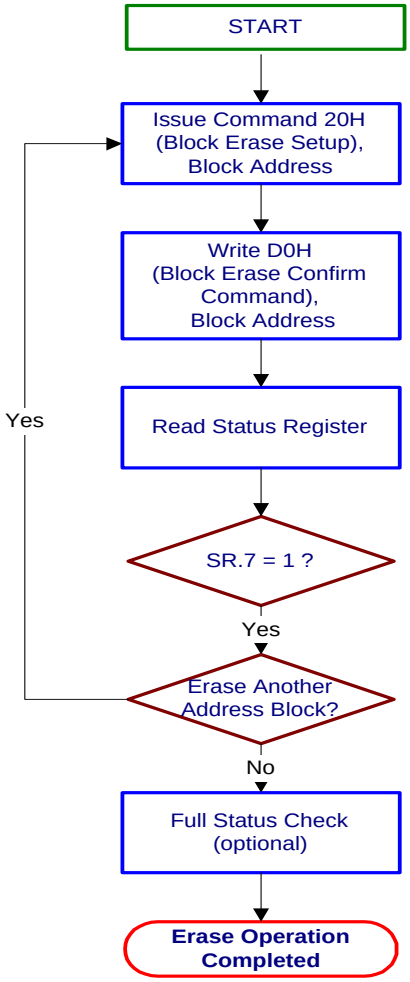
•If SR.7 = "1", the Write State Machine is ready to program more data

•If SR.7 = "0", the Write State Machine is busy.

The status register check is cumulative, which means that if any program operation fails, an error occurs and the program operation is aborted. Thus, if 100,000 data locations have been programmed with one failing cell, the results will be reflected in the data register.

1.4.3 Intel® Block Erase Algorithm

Any non-blank flash device must be erased before it can be programmed. The Intel® Block Erase Algorithm performs and verifies device erasure. Block erasure is initiated by a two-cycle command sequence. To erase a block, issue the *Erase Setup* command (20H) and the *Erase Confirm* command (D0H) to the Command User Interface, along with the address of the block to be erased. Performing a block erase sets all bits within the block to "1". Only a single address block at a time can be erased. The flow chart illustrates how the Intel® Block Erase Algorithm works.



1. A two-cycle command sequence initiates the block erase procedure. First, write "20H", the *Erase Setup* command, to the CUI along with the address within the block to be erased.

2. Next, write "D0H", the *Erase Confirm* command, to the CUI, along with the address within the block to be erased. The erase operation does not begin until an Erase Confirm command has been issued. After this command is issued, the Write State Machine executes the following events within the device:

   - Programs all bits within the block to "0".
   - Verifies that all bits within the block are programmed.
   - Erases all bits within the block by changing to "1".
   - Verifies that all bits with the block are erased.

3. The Write State Machine Status pin, SR.7, is checked to determine if erasure is completed.

- If SR.7 = "1", the Write State Machine is ready and erasure is completed.
- If SR.7 = "0", the Write State Machine is busy.

4. If more address blocks are to be erased, repeat steps 1 to 3 until erase operation is completed.

1.5 AMD® Algorithms

For AMD® compatible flash devices, the *command register* serves as the interface between a flash device and the microprocessor. An internal state machine uses the command register input to control device erasure and programming. Read and write device bus operations are initiated by writing commands to the command register. Device programming and erasure occur when the appropriate command sequence is written to the command register. For example, the *program command sequence* initiates the **Embedded Program** algorithm, which automatically performs device programming and verification functions. The *erase command sequence* initiates the **Embedded Erase** algorithm, which automatically preprograms the array and executes the erase operation. In order to write a command sequence for programming data or erasing sector addresses, the system must drive WE# (the Write Enable pin) and CE# (the Chip Enable pin) to low, and OE# (the Output Enable pin) to high. The Embedded Program algorithm and the Embedded Erase algorithm are presented in the following sections.

1.5.1 The AMD® Embedded Program Algorithm

AMD® compatible flash devices are programmed using the *program command sequence* to initiate the **Embedded Program** algorithm. The Embedded Program algorithm causes the device to automatically perform programming and program verification functions. To determine if the program operation is completed, the system reads the Data Polling bit (DQ7) or checks the Ready/Busy (RY/BY#) status pin. If the RY/BY# pin equals "1" or DQ7 contains the data programmed to DQ7, programming has been completed. The device is then ready to accept another command or read data.

Programming is a four-bus-cycle operation initiated by the *byte program command sequence*. The corresponding bus-cycle operations are listed below:

- First bus-cycle: unlock write cycle
- Second bus-cycle: unlock write cycle
- Third bus-cycle: program set-up command
- Fourth bus-cycle: write to memory address location and write data at specified address

Addresses are latched on the falling edge of **WE #** or **CE #**, whichever happens last in the bus cycle. Data is latched on the rising edge of WE # or CE #, whichever happens first in the bus cycle. The rising edge of WE # or CE # begins the programming operation

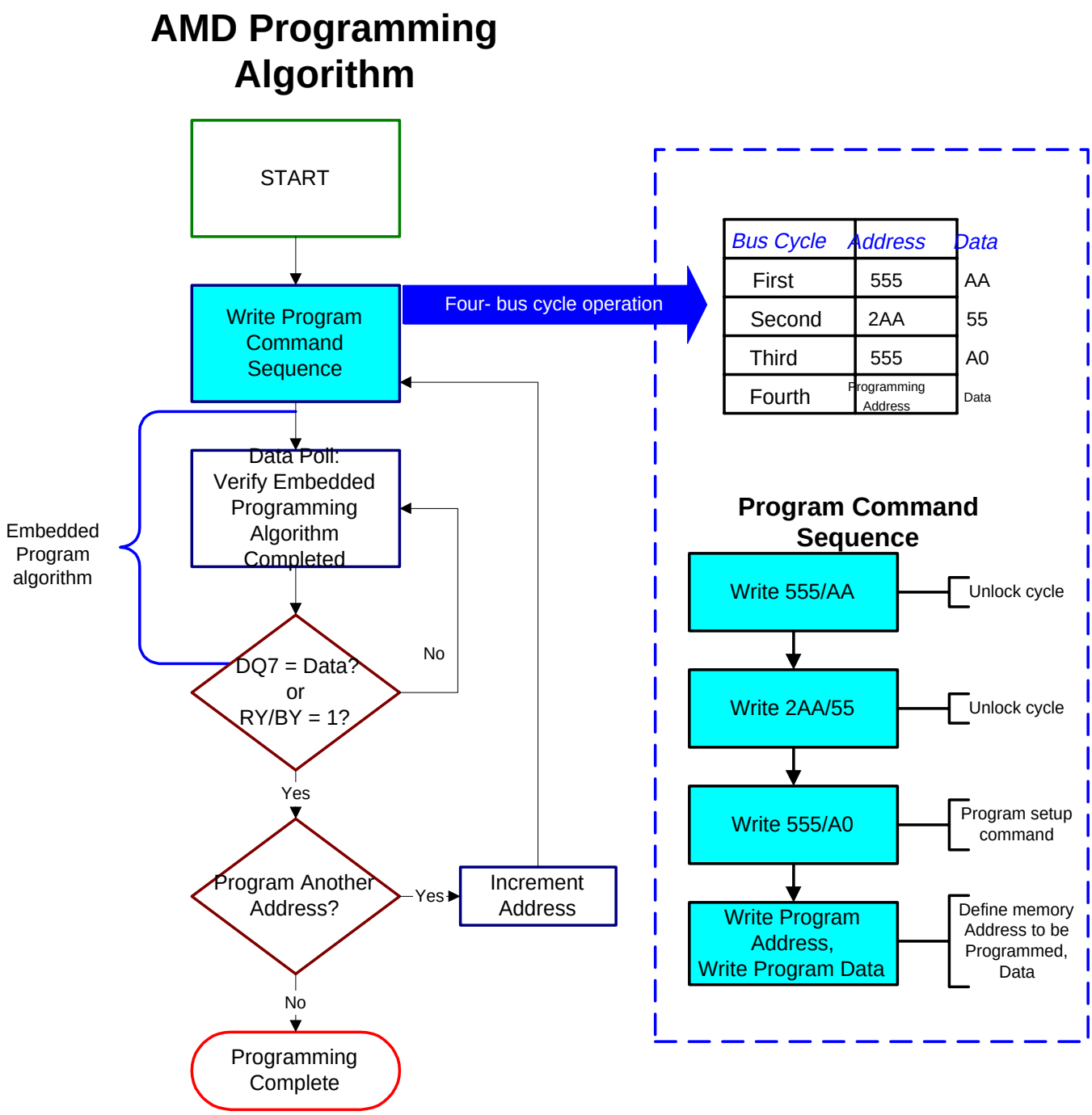


Figure 1-1 AMD® Programming Algorithm

The AMD® Programming Algorithm flowchart shows the following sequence of commands and events:

1. The writing of unlock commands such as 5555H/AAH, 2AAAH/55H and the program command 5555H/A0H is part of the command sequence that sets up the AMD® flash device for programming. The address and data to be programmed are written to the device. Pin "DQ7" goes to the opposite of its expected state while programming.

2. The automatic programming operation is completed when the Data Polling on pin "DQ7" is equal to the data written to this bit.

   The RY/BY# pin indicates to the host system that the embedded algorithms are either in progress or completed. If the RY/BY# pin read is low, the device is busy with the program operation. If the output is high, the device is ready to accept additional read/write operations.

3. After a sector is completely programmed, the address is incremented and new data is programmed.

1.5.2 The AMD® Embedded Erase Algorithm

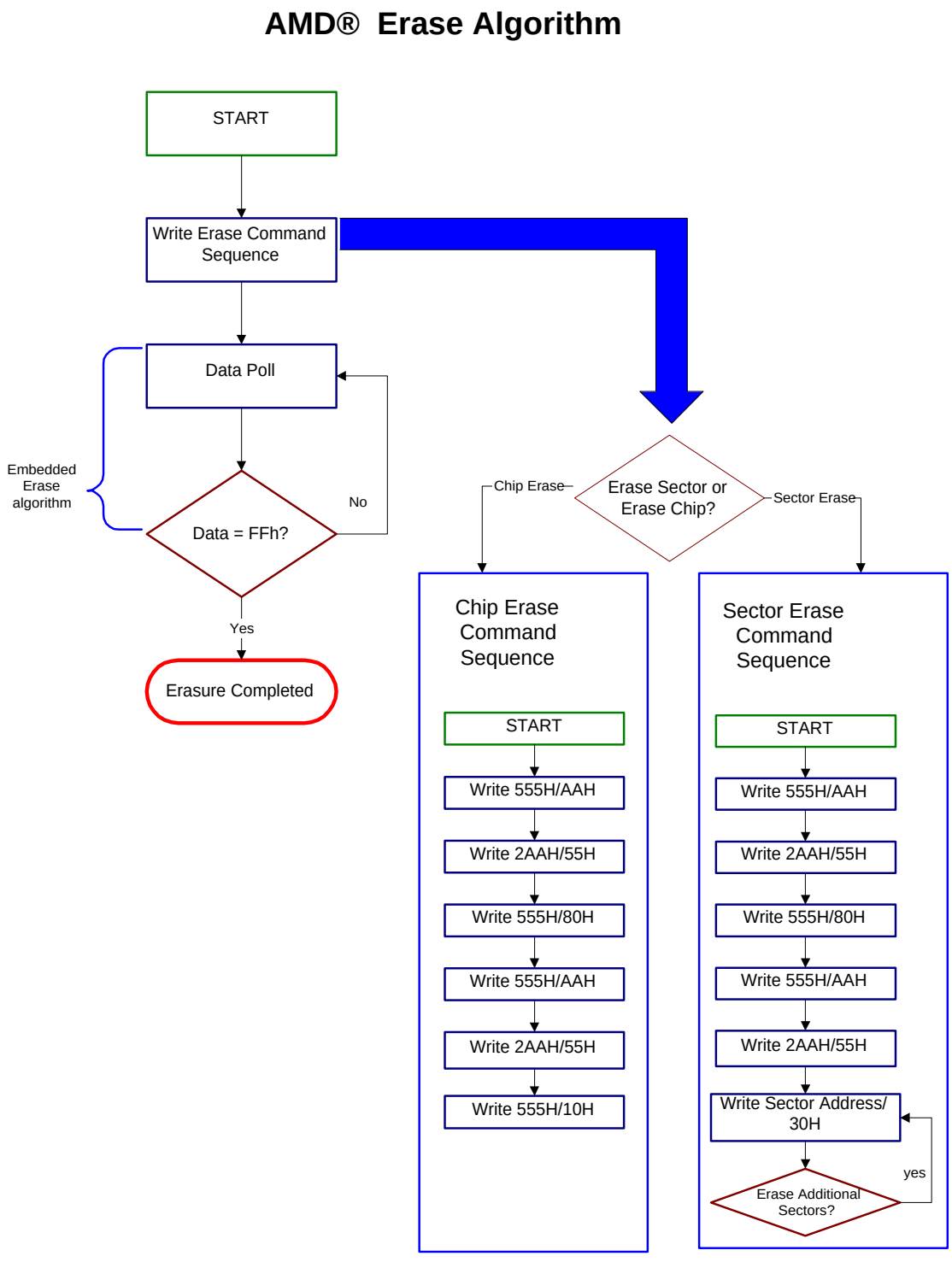


Figure 1-2. AMD® Erase Algorithm

The erase command sequence is used to erase AMD® compatible flash devices. You can erase a single sector, multiple sectors, or the entire device. There are two common erase algorithms used for AMD® flash devices, Chip Erase, and Sector Erase. Each algorithm uses a six bus cycle operation. The following sequence of commands and events occurs with this algorithm:

- The erase sequences are initiated with two "unlock" bus cycles. The unlock cycles provide data protection against inadvertent writes.
- A "set-up" command is written to the command register.
- Two more "unlock" write cycles occur.
- **Chip Erase:** the *chip erase command* is written which triggers the Embedded Erase algorithm; or
- **Sector Erase**: the sector erase command consists of the address of the sector to be erased followed by the sector erase command 30H. This command can be repeated to erase multiple sectors.
- The Embedded Erase algorithm automatically pre-programs the entire memory or programs the sector and verifies that an all zero data pattern exists prior to erasure.

1.6 The Advantages and Limitations of On-board Programming

HP 3070 Series 3 hardware and software allows you to program flash memory devices at in-circuit test. There are several advantages and limitations to consider when determining whether to use on-board programming to program flash devices.

1.6.1 The Advantages of On-board Programming

✓ Cost reduction opportunities exist:

- ECOs are simpler to implement because flash programming can take as little as a day to set up. Once set up, code can be changed to accommodate new versions in minutes.
- There is no need for off-line programming. This can save between $.40 and $.60 per part.

- When the programming time of in-circuit tests (ICTs) is less than allotted throughput time for board test, the cost of **ICT** on-board programming is free.

✓ Reduced handling can decrease the percentage of damaged parts:

- Flash devices are soldered to the PCB using surface mount assembly equipment. Reduced manual handling of components minimizes the potential for damaged or misaligned component pins.
- Additional handling equipment is not required for Small Outline Packages (SOPs).
- Automated component handling results in fewer bent component leads.
- Tape and reel media can be utilized for flash memory component installation.

✓ Hardware costs can decrease:

- There is no need to add flash memory sockets to the test boards.
- Combining flash programming with the in-circuit test process decreases total manufacturing expense.

✓ Inventory control costs will decrease significantly:

- On-board programming of flash memory eliminates the need to assign additional part numbers to programmed devices.
- No individual device labeling is necessary because automated test equipment reads the part identification off the device.
- No inventory storage is required for programmed flash devices.

1.6.2 Limitations of On-board Programming:

Designing for OBP can increase hardware costs slightly:

- Additional circuitry that can three-state outputs of any components connected to flash devices is recommended to prevent damage to parts from excessive backdriving.
- Additional circuitry is recommended to prevent signal conflicts and bus contention.
- A full compliment of test land pads are needed for **ATE** programming unless slower methods such as **JTAG** Test Access Ports are used.

Test evaluation methods may need revision:

- Manufacturing line test throughput time increases by the amount of time it takes to program flash devices. Flash programming time varies from five to sixty seconds.

Board test developers assume a more significant role in the early stages of projects.

- Since functionality is added to boards by programming flash devices at test time, board designers should plan for the testing process by communicating closely with test developers. This modifies the quality assurance role traditionally performed by test developers to more of a developmental role in the board development cycle.
- Faulty ICT program tests cannot be skipped without manufacturing process modifications.

Chapter 2
Design For On-board Programming

Rev. A

This chapter requires an understanding of the flash programming concepts described in:

- **Chapter 1, “Introduction to Flash Programming”**

This chapter describes:

2.1 On Board Programming

On-board programming utilizes automated test equipment to program flash memory devices that are installed on a printed circuit board. HP 3070 system hardware and software can be used to program flash devices on-board. Programming flash devices on-board in the production phase of board test development involves tasks almost identical to those currently utilized by HP 3070 test developers. However there are additional design consideration that should be incorporated into an effective on-board programming strategy. After learning some on-board programming fundamentals, programming flash devices in-circuit is no more complex than developing tests for other devices.

To simplify flash test development, HP 3070 systems provide digital libraries for many common flash devices. These digital libraries use standard Vector Control Language (VCL). If an exact match for your device is not found in the library, it is easy to create a custom test. Simply search the libraries for flash devices similar to those installed on your test board and make minor modifications to the library test. Then the standard flash library will work with the flash device under test.

NOTE

Although not required to perform flash programming, **Flash70** can be utilized to dramatically increase the speed of flash programming.

2.1.1 OBP: A Different Approach to Test Development

Successful OBP requires a different approach to test development. Traditionally, the testing phase of the board development process has played a quality assurance role in manufacturing. When programming flash devices on-board, the testing phase assumes a more significant role in the developmental stages of projects. For instance, software version control is managed by the test developer and board designers need to ensure that the board meets the test developers' requirements for OBP. Also, functionality is added to flash devices during production.

With on-board programming, design-for-testability should be emphasized. For best results, board designers should work collaboratively with test developers to incorporate testing conditions into the board design. Implementing OBP of flash memory with a design-for-testability approach can ultimately increase the profitability of most board design projects.

2.2 Planning for Flash On-board Programming

Planning is essential for quickly and easily implementing flash programming at in-circuit test time. Special design criteria for OBP should be added to the standard design criteria. It is necessary to plan board designs that allow the safe programming of devices and to provide sufficient information about data structures and data sources to the test programmers. To attain the maximum value from flash technology, design teams should create boards that can be programmed in-circuit, on-board, or by an in-circuit test, if required.

2.2.1 On-board Programming Design Considerations

The following design conditions are recommended for flash on-board programming:

- **Adequate probe access** to all flash device pins is needed, especially if 16 bit data is used for OBP.
- Circuitry that **prevents signal conflicts or bus contention** during OBP should be incorporated into the board design.
- An **on-board voltage regulator** should be used to ensure that V_{PP} and V_{CC} voltages used at test development work within the thresholds of in-system voltage specifications.

NOTE

Inaccurate programming can be caused by voltage differences between flash device V_{PP} requirements and the on-board power supply. Provide flash device specifications to board designers so that this problem can be addressed in the board design process.

- Designing a board so an on-board processor can program flash memories does not guarantee that the board can be programmed successfully in **ICT**.

  If the flash device is being driven by another in-circuit device, it is unlikely that the **ATE** will program the device properly since the processor programs the part.

  ICT usually backdrives output from upstream devices to perform cluster tests. The HP 3070 protects these devices against excessive backdriving with the Safeguard Protection feature. However, using the safeguard feature for flash OBP slows the process down to an unacceptable speed. To

effectively program flash devices, the HP 3070 safeguard feature must be turned off.

- The ability to three-state upstream devices is essential for **OBP**. The only way to protect upstream devices is to incorporate three-stating mechanisms into the board design.

  It takes between 2 and 45 seconds to program a flash device. Some studies have shown that devices can typically withstand backdriving for only a few milliseconds before damage results. Therefore, it is necessary to turn off the "safeguard" feature of the HP 3070 system. Since flash programming cannot be achieved in an acceptable time with safeguard on, three-stating devices on the board is the correct solution.

2.2.2 Board Design Recommendations

2.2.2.1 Disable Bi-directional Signals to Prevent Bus Conflicts

For digital testing, the capability to disable other devices on the bi-directional signals of the device under test is critical. The HP 3070 cannot program flash devices in-circuit unless all bi-directional pins on the data bus are disabled. Therefore, an important element of **OBP** design is the capability to quickly and easily disable the board's bi-directional signals with the automated test equipment. The best OBP board design utilizes a single input that can disable all parts directly accessing the memory to be programmed. This strategy enables more error free programming results. If this is not possible, it is important to supply test developers with disabling specifications for all ASICS and custom devices.

2.2.2.2 Disable Input Signals to Prevent Backdriving Damage

Newer devices are more resilient to overdriving damage than those produced in earlier generations. Also, new HP 3070 programming libraries release backdriving signals more frequently within the test to limit backdriving time. However, board designers should provide isolation of all I/O pins on the device to be programmed to protect upstream devices from damage that might result from long periods of backdriving activity.

Disabling can be accomplished by three-stating devices. There are many methods to three-state devices for flash programming. Some of the most common **OBP** three-stating methods follow:

- Design ASICs with fully three-statable outputs. This eliminates the need for tests to overdrive the device's signals.
- Use volatile FPGAs, which power-up in a three-state condition.
- Add three-statable buffers to protect output. Commercial mpu are often difficult to disable. If the test developer disables mpu, the output signals are not likely to be three-stated along with the bidirectional signals.
- Use HP Boundary-Scan Interconnect Plus to three-state signals on **BSDL** compliant devices. HP Boundary-Scan Interconnect Plus automatically disables all I/Os on the device.

2.2.2.3 Provide Access to All I/O Signals

Some board designs use only 8 bits of data on flash devices that can be operated in 8 or 16 data bit mode. Since flash devices can be programmed by word-wide methods, it is much more efficient to retain access to all device signals for ICT. In 16-bit cases, if the board design permits full access to all pins on the flash device, programming time can be reduced by almost 50 percent.

2.2.2.4 Use System Power Supply Levels and Document Operational V_{cc}

Power and ground transient noise spikes are another source of OBP power supply problems. Board designers must address power, ground noise, and signal integrity issues in **OBP** environments. Pin drivers often route signals across long distances through **ATE** equipment which adds capacitance and inductance to the transmission line. By the time the programming voltage reaches the flash device under test, the signal integrity can degrade.

Utilize in-system power supply levels to gain the most reliable OBP results. This reduces the risk of inconsistent voltage applications, because the automatic write mechanism of some flash devices verifies the data content against thresholds relevant to the V_{CC} levels. If the V_{CC} level during programming is lower than it will be in the final product, data bits which verified as high during program verification may read below the high threshold during product operation.

If there is a voltage regulator present to generate V_{CC} during testing, then discrepancies in power supply levels disappear. When V_{CC} signals are provided directly from the HP 3070, the design team should document the operational V_{CC} level for the test team.

2.2.2.5 Establish Direct Access to BSDL Signals

Devices that are compliant with IEEE 1149.1 standards can take advantage of Boundary-Scan technology to provide disabling. Boundary-Scan automates the disabling process via **BSDL**. Because most testing and programming of programmable logic is performed through the boundary-scan ports, having direct access to these signals and providing chains to interconnect them enhances the testability of the board. Designers need to supply accurate boundary-scan description files that operate properly with the devices on the board. The importance of Boundary-Scan for testing should be emphasized in on-board programming design plans.

2.2.2.6 Provide Data Protection and Disabling Information

Most flash devices have programmatic protection capabilities against V_{PP} voltage so the board design does not need to be modified for **OBP**. Some older flash devices have data protection features which require that 12v be applied to address pins for erase and program operations. However, if the test program needs to apply 12 volts to an address line, buffers should be added to isolate V_{PP} from the address line to protect upstream devices from V_{PP} damage. When using this method, it is very important that the designer provides information to the manufacturing test team about how to disable access of the V_{CC} address lines when the program protection voltages are in use.

2.3 What Test Developers Need to Know

To generate successful flash programming tests, we recommend that test developers establish the following goals:

- Prepare a testing specification document for the designers of the board.
- Learn how to program devices with various types of data.
- Learn to verify that data is correctly programmed.
- Clear the data bus of all activity from other devices during programming. This is the most important element of the flash testing process with ATE.

  In digital tests, interference from other devices can result in programming intermittence. Therefore, complete and easy disabling of upstream devices is essential for effective and safe **OBP**.

NOTE 

When upstream devices interfere with the programming of flash devices, the programming test may not fail. The data being programmed, however, will be incorrect. Adequate disabling prevents this type of problem.

- Three-state all upstream devices which are exposed to backdriving. Although the "safeguard" feature of the HP 3070 provides protection from backdriving, safeguard must be turned off to achieve optimal programming times.

2.3.1 What is a Flash Programming Test?

Flash on-board programming tests are digital tests that use part libraries to actually program data onto an in-circuit flash device. Flash programming tests are simply digital "pin library" tests written in VCL. Typically, IPG generates six flash library tests during test development. The two tests required for flash **OBP** are "erase" and "program" (see **Flash Programming Tasks, on page 1-4)**.

Flash digital libraries contain information such as flash programming voltages (Vpp), in-system voltages (Vcc), and pin assignments. The **PDL** files included in the HP3070 B.3.00 software point to digital tests that program the most common flash devices on the market. The flash digital library models used for **OBP** testing are described more

completely in **IPG, PDLs, and Flash Test Library Models, on page 6-11.**

2.3.2 Data Sources and Board Topologies Effect OBP

Choosing the best method to program flash devices in-circuit depends on several factors, including intended data sources and board topologies. It is important to understand the data sources provided and to make efficient use of them on various board topologies. Because flash devices and board topologies differ from project to project, knowledge of data structures and board topology is a very important **OBP** consideration.

For programming flash memory devices, HP 3070 systems support Intel® Hex, Motorola® S-Records, and decimal data source formats. Each flash device to be programmed requires a data source file. Understanding how to interpret these data formats enables you to compare the actual programmed data on a flash device to the expected data results.

For effective on-board programming design, it is important to understand the types of board topologies that enable faster programming methods. We recommend that board designers prepare documentation that defines the board topology and data source structures. Test developers can then use this documentation to develop effective test strategies.

2.3.3 Board Topologies for On-board Programming

2.3.3.1 Individual Flash Devices Connected by Separate Data Busses

Test developers often work with boards that use separate data busses for each on-board flash device. This type of flash test is easy to implement because board designers typically provide one data file for programming. With one data file, the source usually matches the bit width of the flash devices on-board.

A diagram of this scenario follows:

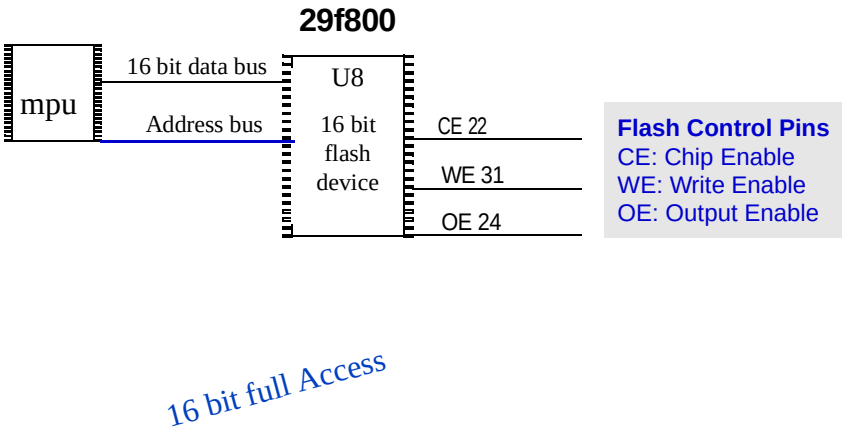


Figure 2-1 OBP on 16 bit board topology

2.3.3.2 A Series of Flash Devices Connected to a Single Data Bus

Sometimes boards have many flash devices connected to a single data bus. Since all parts utilize the same data nodes, it is necessary to program the flash devices sequentially. With this type of design, **ATE** probes must have node access to the chip enable pins on all devices connected to the data bus. This enables each flash device to disable the others.

With this type of topology, the test program must ensure that the inputs to the other flash memories are disabled. In the topology depicted below, WE or OE lines are held in common. Each device has an independent chip enable. This means disabling can be automatically implemented by the HP 3070. The disable subroutine of the program test disables the appropriate parts.

The topology depicted below represents a series of flash devices connected by a single data bus.:

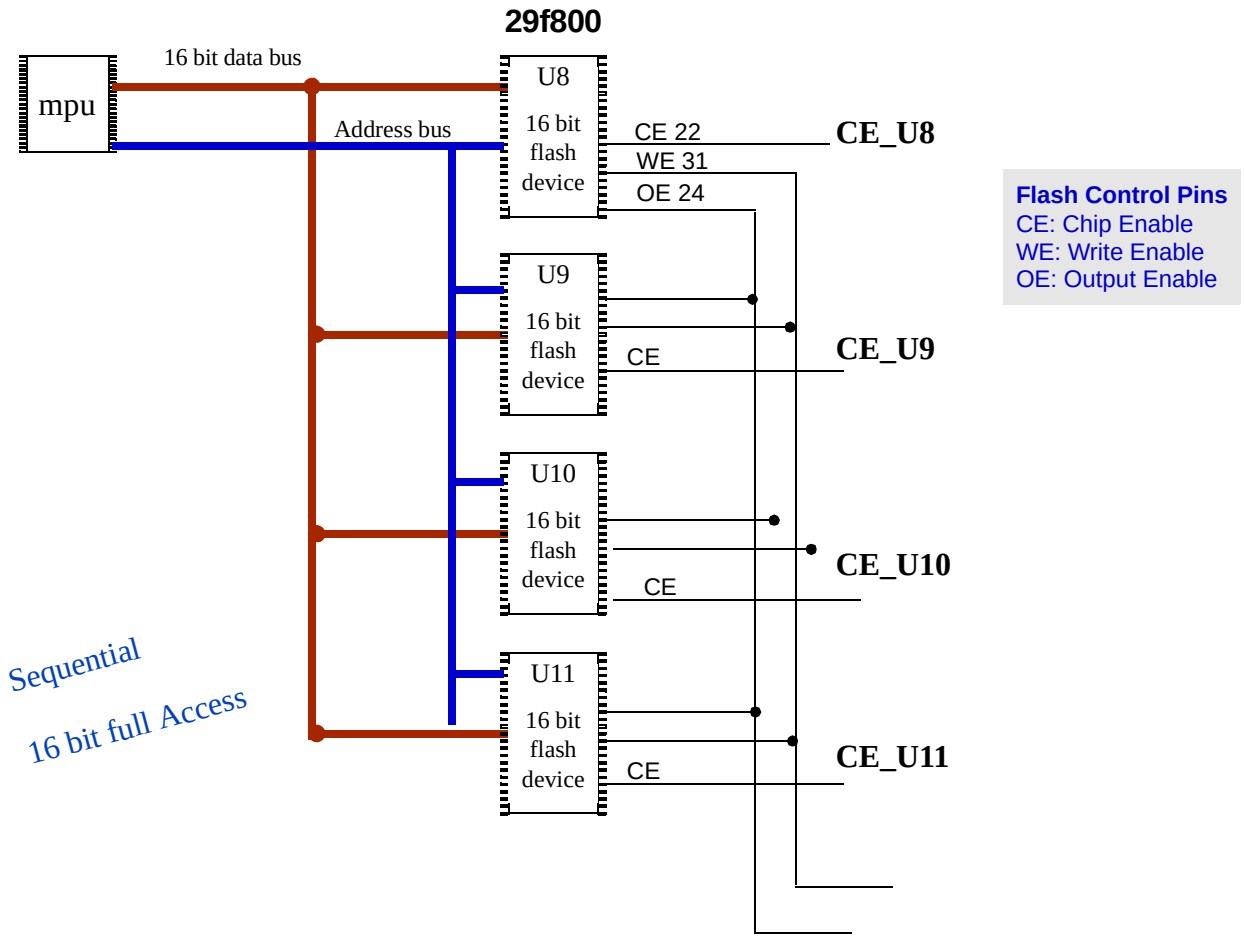


Figure 2-2 Multiple Devices and Single Data BusTopology

2.3.3.3 Multiple Flash Devices Connected to a Single Large Data Bus

Some board designs connect multiple flash devices to one large data bus. The example below shows a 64 bit mpu and four flash devices connected to separate data busses. This design is especially good for **OBP**, because disabling of other flash devices is not necessary. However, any peripheral devices connected to the flash device data bus must be disabled. Also, the HP 3070 is capable of programming all devices in parallel with a single cluster test to improve overall programming speed.

In this type of design, when separate nodes must be asserted to three-state upstream devices, these must be added to the program test.

A diagram of the topology follows:

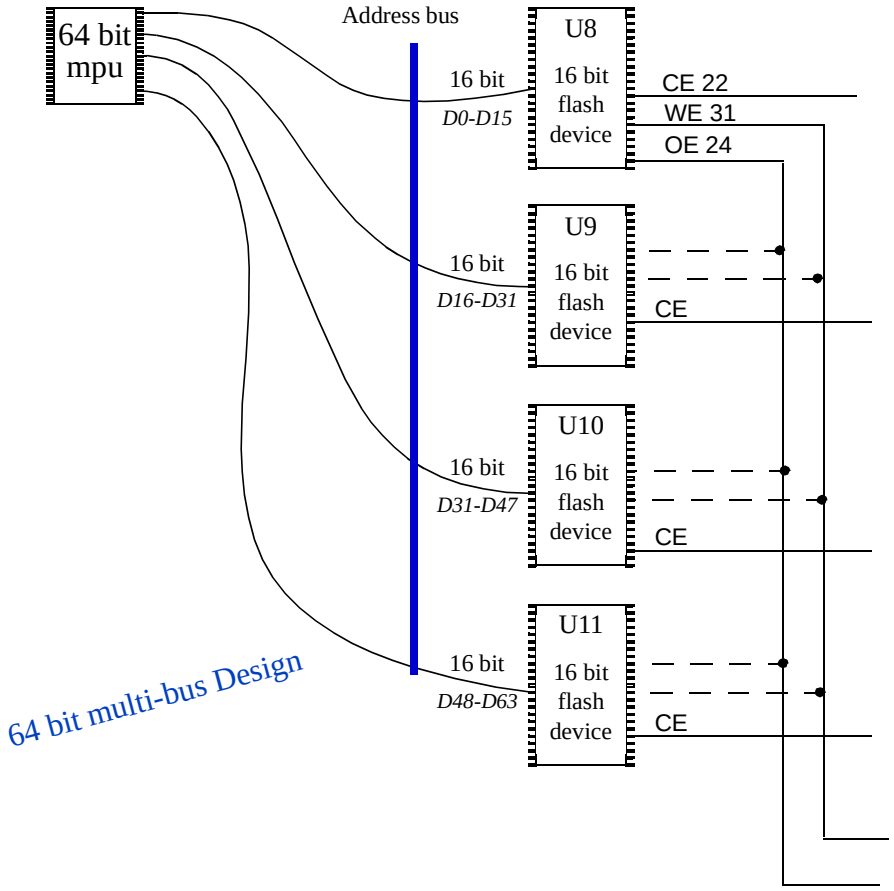


Figure 2-3 Multiple Flash Devices on a Single Data Bus

2.3.3.4 Parallel Flash Programming With HP Throughput Multiplier

HP 3070 Throughput Multiplier is a software tool that can be used to perform Flash **OBP** parallel programming of identical boards. HP Throughput Multiplier allows you to program two or more boards of the same type and executes flash tests simultaneously on each board.

A diagram of this topology follows:

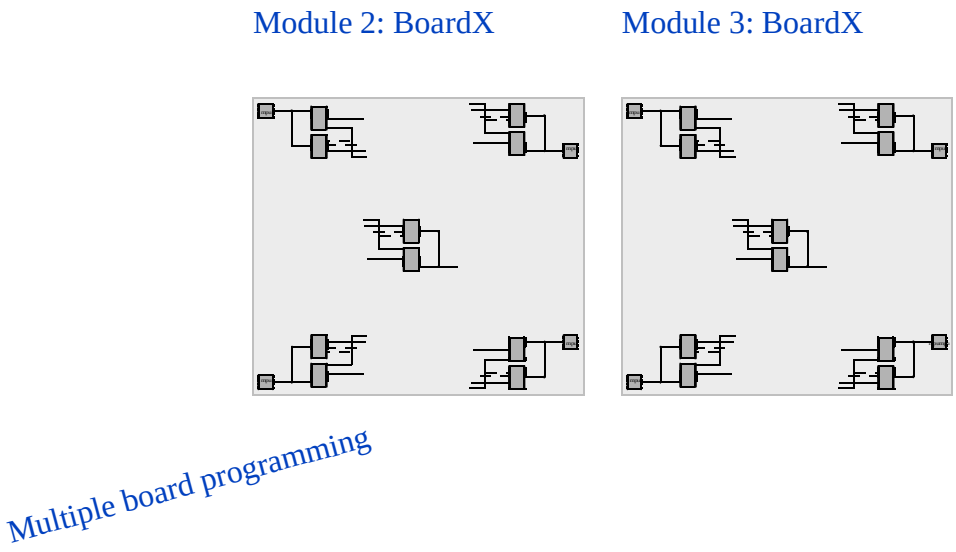


Figure 2-4 Flash OBP Parallell Programming Topology

2.3.4 Creating a Sample Design Document

We recommend that board designers and test developers create a flash programming design specification document. This document should include information about the flash devices to be programmed, board topology, address buses, data buses, and disabling information. An sample design document is shown next.

Date:

Design Engineer: _____ Phone #: _____

Project Name: _____ FAX #: _____

Project Description: _____

_____

_____

_____

Flash Device Type: Part Number:

Size: _____ Quantity: _____

Pin Topology: _____

Board Topology Diagram (flash devices, data bus, address bus, control lines) _____

If multiple devices per board, show addressing and data locations based on the full bus of the board:

Address Bus				
Data Bus				

Are some data locations to be programmed based on individual board version or serial number? _____

How often will software versions be updated? _____

If yes, what locations? _____

Will this board have software versions? _____ If so, on what basis will they be programmed? _____

How often will software versions be updated? _____

When will data be finalized? _____

List the upstream devices that are fully three-statable and the method used: _____

Where are the board disable lines? _____

Discuss other disabling issues: _____

Chapter 3
Flash70 Digital Tests

Rev. A

This chapter describes:

3.1 What is a Flash Digital Test?

A flash test is a standard VCL digital test that uses an external data source to program a flash device. Unlike standard digital tests in which the data values are explicitly defined by internal vectors, flash digital tests program unpredictable data values from an external file. The external file is a formatted data record which provides the data the compiler uses to program the flash device. A programmer defined data structure called a data block specifies the external data source used to program the device. **VCL** statements within the data block define the external data file and how its data is interpreted. Dynamic vectors extract data and address information from the specified data file. The address and data are then applied to the device to be programmed within a data cycle created by standard VCL vectors. The compiler interprets the file and calculates the appropriate address to be applied with each byte of data.

Another difference between a standard digital test and a flash programming test is size. Flash tests are usually many times larger than standard digital tests. To accommodate the large size of flash device tests, flash data is programmed in smaller sections called segments which the HP 3070 treats as multiple test files. Any size data file can be programmed by one digital test by using a repeat loop that maximizes pin RAM, and directory and sequencer RAM usage for programming.

3.2 The Series 3 Flash Compiler

The Series 3 flash compiler is more efficient than earlier versions and works with Motorola® S-Records, Intel® Hex records, and integer data. Since most designers work with cross-compilers that generate Motorola® S-Records or Intel® Hex records, many new Flash70 features are optimized for these types of formatted data files.

3.2.1 Data Interpretation

The compiler parses the data and address sections of the formatted data records based on the definitions provided in the data block. Motorola® and Intel® record types contain a series of ASCII values representing bytes of data. How the compiler interprets the data from a formatted record depends on the cross-compiler used to generate the files and the device or devices to be programmed. Several user definable options are available to manage and define variances in data interpretation.

Options that can be defined within the data block include a **step modifier** and a **format modifier**. A step modifier is used to translate the address from the data file into an address that fits the pins of the device under test, as well as the board topology. Some cross-compilers and data file generators create data records in which the address is not incremented by each byte, but by each 16 bits of data. Thus, 16 or 32 bits of data are treated as one address location, rather than the expected single address per byte. The format modifier can be used to handle non-byte addressing in Intel® Hex and Motorola® S-Records.

3.2.2 Automatic Segment Removal

When the number of repeat loops or programming sequences is larger than the data source available, the compiler interprets data more efficiently. If the data is exhausted, the compiler does not execute unnecessary segments[1]. Segments must be programmed in their entirety. Thus, when data runs out before a segment has been completed, the compiler programs reverts to an ***end-of-data*** condition and programs "harmless" data, FF, to a single location. By default, this is the highest address in the device. Since this may not be the appropriate address for the data, the test programmer can define information in the data block to select the appropriate address location for the data to be programmed. Programming FF can result in many unnecessary programming sequences, increasing flash programming execution time. If FF data is distributed throughout a data file, the Series 3 compiler automatically removes these blocks of FF data. Several seconds may be saved by not programming this redundant data.

3.3 Flash70

Flash70 is an optional software product that improves flash programming time. Flash70 features include:

- Faster programming speed with the Flash70 algorithm:
- New library models to simplify flash test development.
- Comprehensive online documentation: the *Flash Programming Guide*

1. This may result in improved test times, depending on the size of the data source and the device to be programmed.

The Flash70 programming algorithm takes advantage of the new Control XT card's expanded memory. If you are not using Flash70, segment size is limited by the vector RAM behind the pins, which can never exceed 8k. With Flash70, segment size is determined by sequence RAM which is 1 MB on ControlXT cards. This larger size RAM increases the potential segment size for programming which improves flash programming speed.

3.3.1 The Flash70 Algorithm

The Flash70 algorithm improves programming speeds for flash devices. Significant improvements in Flash70 test speed are the result of decreased overhead for segment execution and more efficient use of expanded ControlXT memory.

To obtain the benefits of Flash70, you must purchase and enable Flash70 software. To enable the Flash70 algorithm, add the statement "enable flash70" in the board "config" file and include the keyword, "flash" in the VCL tests for your flash devices.

When Flash70 is used in combination with the flash compiler, dynamic vectors use the Flash70 algorithm for programming operations. This algorithm uses the faster, larger memory of the control card for all the variable data pointers. The slower pin card memory is used for data that doesn't need to be reloaded during the test. Thus, fewer segments are required to program a device.

3.3.2 Faster Tests with the Flash70 Algorithm

NOTE

It is not necessary to know the internal details the Flash70 algorithm to program flash devices. Read this section if you want to learn why the Flash70 algorithm is faster.

Flash70 utilizes an algorithm that improves the speed of programming flash devices. This improvement is the result of vector expansion. Vector expansion differs from the standard flash algorithm in the following way. With the standard flash compiler, pin RAM is used to store the combinations of ones and zeros needed to program data. Since pin RAM is only 8k, the time required to reload data becomes detrimental to flash programming performance. For example, a 35 pin address and data bus has 2^{35} possible binary combinations to hold address and data information. Potentially, up to 34 Gigabits of information are needed to program the device. Theoretically, if the data file is perfectly random, the pin RAM must

be reloaded 4,000 times to program all the combinations (34Gb/8k = 4,000 Pin RAM reloads).

The Flash70 algorithm eliminates the time consuming need to reload pin RAM. Flash70 vector expansion divides the data into more easily managed chunks by programming eight to ten pins at a time. The pin RAM required to contain every combination for eight to ten pins is only 2^8 to 2^{10} bits (i.e. 256 to 1024 combinations). Since the required memory is less than the 8K of pin RAM available, programming a device becomes a matter of loading the Pin RAM once with every combination possible for the smaller sections, and then using ControlXT Sequence RAM to organize the data.

The Flash70 algorithm expands one vector into a multiple vector execution sequence. Some data records may require several vectors to implement programming completely, since multiple vectors are executed for one dynamic vector. The execution rate for the subvectors is 80ns per vector. In the example below, a single 160ns dynamic vector will become four 80ns vectors. The dynamic vector completes its cycle in 320ns while all other vectors operate at the user-selected vector cycle.

To better understand vector expansion, consider the following, drive only, example:

Data to be Programmed:

```
Address_19 "000 0000 1010 1111 0101" Data_16 "1001 1100 0100 0010"
```

The Execution Statement:

```
Execute Keep_control drive data Address_19 drive data Data _16
```

The execution statement above expands the address information to multiple vectors. *Keep_control* specifies that the state of some of the pins from the previous sector remains unchanged. Each vector cycle changes only part of the data on the bus (between 8 to 10 bits). On the final vector cycle, the flash device receives the complete record.

Flash70 creates a four vector execution sequence from the single data record. In the example above, the execution sequence expands in the following order:

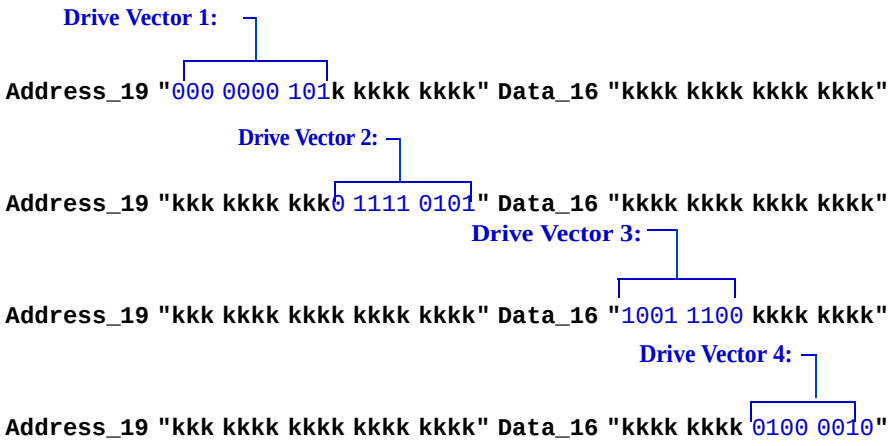


Figure 3-1 Flash70 Dynamic Vector Expansion

The Flash70 algorithm programs the entire device by transferring data records section by section to the data bus in the following order:

1. Drive Vector 1: 10 high order address bits.
2. Drive Vector 2: 9 low order address bits.
3. Drive Vector 3: 8 high order data bits.
4. Drive Vector 4: 8 low order data bits.
5. On the final write, **WE#** is asserted to program the entire address and data information to the flash device.

For example, WE# or **OE#** are offset from the expanded vector in which they are active, proportional to the user vectors. Final control lines are driven in the last vector to meet DUT requirements. Since there are no receives in this statement, the 6 MHz hybrid card receive limitations are not encountered.

Data read sequences depend on the multiple vector algorithm to create the vectors which comply with the 6 MHz hybrid card limitations. All control lines are driven in the second address vector. The **DUT** has at least one vector to settle its outputs before the receive vector is active. If more than one receive vector is executed, the only read receive is on the last vector.

Other notes concerning the Flash70 algorithm follow:

- The receive delay should always be set to 100ns or less, when using a vector cycle of 160ns. The Flash70 compiler uses offsets for the receive delay. If the delay setting is greater

than 100ns, offsets cannot be used. In this case, the expanded vectors operate at the user defined vector speed.

- RAM behind the pins is loaded with fixed, static data.
- Sequence RAM dynamically controls the sequence of the data applied to pin RAM.
- The number of data and address pins on the data bus determines the quantity of vectors expanded. This process is controlled by the sequence RAM.

3.3.3 Obtaining 12MHz Speed on 6MHz cards

With Flash70, if you have double density 6 MHz hybrid cards in your system, it is possible to achieve additional improvement in programming speed. Flash70 enables some users to program at rates faster than the 160ns specification indicates. With good fixturing and fast memory, 80ns vector cycle time may be achievable. Since vector execution is very fast in flash programming, the faster cycling can improve programming performance significantly.

The 6 Mhz hybrid cards are unable to drive signals to a device and receive output is less than 160ns. The new Series 3 flash libraries are designed to drive and receive on different vectors to avoid violating the 6Mhz hybrid card specifications. The Flash70 compiler allows up to 80ns vector cycles when you override the board configuration to optimize overall programming times, even on 6Mhz cards.

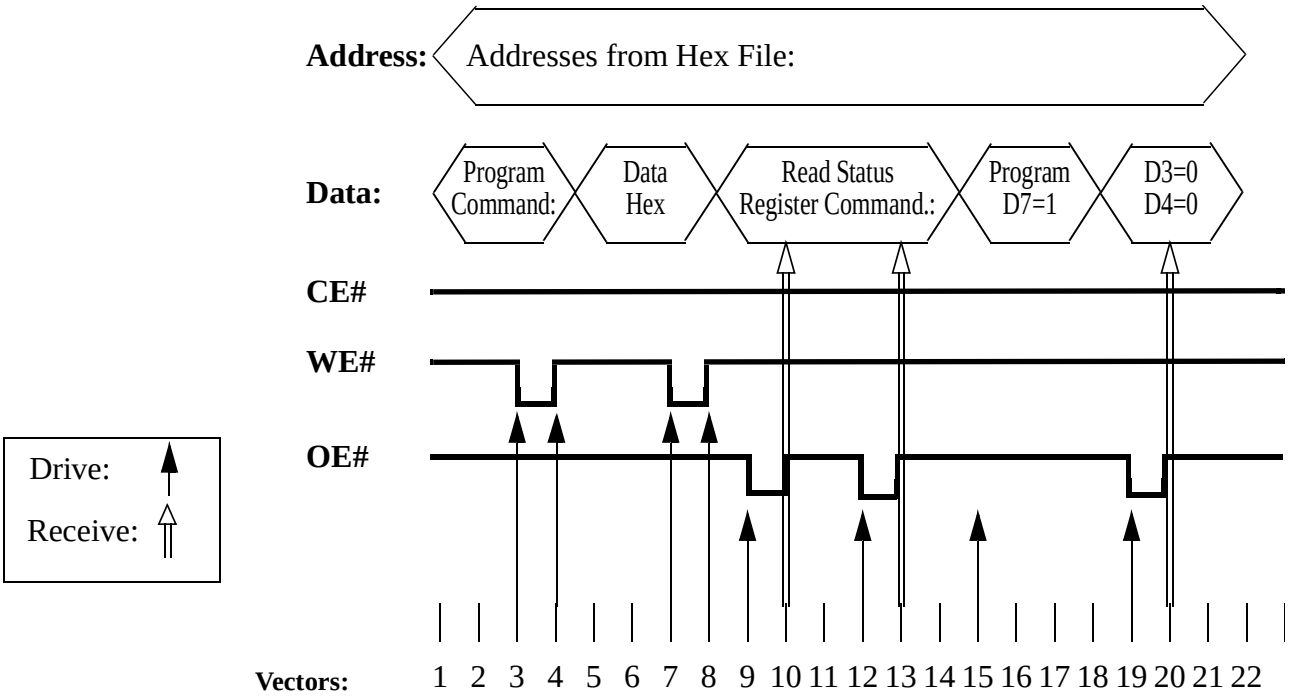


Figure 3-2 Flash Library Vector Cycle

NOTE: This diagram shows a standard programming sequence with the required device bus cycles. Observe the ICT drive and receive requirements. Note that the changes of inputs do not coincide with data receive vectors. This means that the test will never be required to change driven data and received data on the same vector. Since this is the case for programming tests, using 80ns vector cycles on a 6 Mhz system does not violate the specifications. This is why Flash70 allows you to override the system configuration. If you choose to override the system configuration on any test but the "program test", the specification will be violated and the test cannot be expected to operate.

The Flash70 algorithm can operate at up to 80ns vector cycles. Improving programming speed is a matter of trial and error. The flash programming libraries specify a vector cycling time that has been proven in the HP lab environment. This vector time, however, is not guaranteed, because the speed at which VCL tests can program depends heavily on fixturing. Efficient fixturing can reduce the time specified. The fastest vector cycle used cannot be less than 80ns.

To improve the speed of your flash programming, try these procedures:

1. Enable Flash70 in the board configuration file and VCL test

files.

- Enter "enable flash70" in the board "config" file.
- At the beginning of the Declaration section, enter the "flash" statement in the "verify" and "program" test files for the **DUT**.

2. Before attempting to maximize the speed, make sure the device programs as expected based on the library definitions provided.
3. In the VCL test, reduce the vector cycle time to 80ns and the receive delay to less than 80ns.

   For example, in the "program" test for the Intel® 28f160, the following VCL statements appear:

```
vector cycle 160ns
receive delay 100ns
```

   In this case, you would change the vector cycle to 80ns and the receive delay to approximately 50ns.

NOTE 

The receive delay must always be less than the vector cycle time.

4. Compile and run the test on the device to be programmed.

   - ♦ Either the flash device programs successfully, or
   - ♦ If the flash device does not program successfully, increase the vector cycle and receive delay times until it does. In many cases, vector cycles can be faster than 160ns.

NOTE 

In rare cases, the vector cycle specified in the VCL test is too fast for reliable programming. If this occurs, the vector cycle time should be increased beyond the library recommendation.

3.3.4 Hardware Waits

A wait suspends execution of a VCL test and waits for a trigger before resuming. With the hardware wait, the triggering lines and states must be set up at the beginning of the VCL test, and cannot be changed during the test. During the wait, the drivers maintain their current states; however, the DUT clock, if any, continues to run (see *Digital–3*). Timed waits terminate at the end of a specified time. Other waits terminate when the specified triggering states are received.

Flash70 allows you to use hardware waits instead of homingloops in digital tests. Hardware waits are used on flash devices that provide a READY line to specify when a cell is programmed. By using the READY line in conjunction with the hardware wait, the need to poll the device is eliminated. With no need to poll the device, homingloops can be removed from the VCL test, reducing the amount of directory RAM required by the test.

Because you cannot break during a wait, always set a test time at the beginning of the digital test that employs hardware waits. Each segment is treated like a separate test, so the test time applies to each segment. Set the test time to .1 seconds for a segment size of 2048. This allows 50 micro seconds per byte for programming.

Setting hardware waits in VCL tests is easy because the statements already exist in test libraries. To set the hardware wait, comment out the homingloop and uncomment the "wait" statement in the VCL test as shown below:

```
! wait line STS
! wait terminated when STS is "1"

homingloop 60000 times                  ! allow cell to program
 execute Device_Ready exit if pass      ! Program Completed
end homingloop
!execute Device_Ready wait
```

Should be changed to the following:

```
wait line STS

wait terminated when STS is "1"

! homingloop 60000 times                ! allow cell to program
!  execute Device_Ready exit if pass    ! Program Completed
! end homingloop
execute Device_Ready wait
"
```

NOTE

By making the above changes, flash programming speed should improve by approximately 15 percent.

3.4 Data Blocks

A data block enables a set of data to be defined in VCL and assigned to a group of pins. Data blocks are defined in the Vector Definition section of the test. During vector execution, the data is read sequentially and applied as vector states to groups of pins.

The data block defines the data source. The source can be a cross-compiler generated hex records, integer, or a series of ASCII values defined directly within the block. The data block statements specify the data to be programmed and its address location. The data values are driven on to the appropriate pins by use of a drive statement on a host vector. This host vector defines the control lines necessary to create the full bus cycle. Since the data source is generally a standard data format and a fixed size, flash digital tests need to interpret and apply the correct data within the context of a standard digital test.

The compiler introduced at B.03.00 provides many new options that enable digital tests to program flash devices. The automatic

interpretations within the software make it easier to get fast, safe and accurate programming results. Flash70 software accommodates the variations in the types of flash devices and data types used in programming. You can activate Flash70 to program your devices by including the "flash" designation within the flash digital test.

For more information on data blocks, see **Chapter 4, “Data Sources for Flash Programming.”**

Chapter 4
Data Sources for Flash Programming

Rev. A

This chapter describes:

4.1 Overview

Flash device programming requires an understanding of data blocks and formatted data records. Within a flash digital test, test programmers provide data structures known as data blocks to define the data source used to program a flash device. The data source is contained in an external file known as a formatted data record. Formatted data records supply the data used to program flash tests.

4.2 Data Blocks and OBP

A data block enables a set of data to be defined in VCL and assigned to a group of pins. Data blocks must be read sequentially; therefore, they can be used in repeat loops. Two types of data can be defined in a data block:

- Decimal integers: These are saved as 32-bit, twos-complement values that can represent positive or negative decimal values or the decimal equivalents of ASCII characters. These can be defined in the VCL test, or passed to the test in variables or from files.
- Pin states formatted as Intel® Hex Format records or as Motorola® S-Records, passed to the test from files.

Data blocks are defined in the Vector Definition section of the test. During vector execution, the data can be read sequentially and applied as vector states to groups of pins. Except that the data is read sequentially, this is basically the same as applying an array of data to a pin group. The VCL statements associated with data blocks are:

- "data" and "end data" (Vector Definition) — start and end a data block definition
- "values" and "file" (Vector Definition) — define the data in the block
- "next" (Vector Execution) — increment the counter so that the next datum will be read
- "rewind" (Vector Execution) — return the counter to the first datum in the data block

In addition to the statements, two parameters, "drive data" and "receive data", are used by the vector execution statements ("count", "execute" and "preset counter") to drive or receive the next datum from the block.

4.2.1 Using Data Blocks for Flash Programming

Data blocks are used in flash programming to copy address and data information from an external formatted record file to a flash device. External data records such as "Hex_Record_Data" and

"S_Record_Data" provide the formatted data that determines record type, length, offsets, and other data for flash programming.

4.2.2 Data Block Example Using a Motorola S-record

```
data S_Record_Address to groups Address1
  file "S_Records" 180 s record address  ! selects address part of "S_Records"
end data


data S_Record_Data to groups Data1
  file "S_Records" 180 s record data     ! selects data part of "S_Records"
end data
```

Figure 4-1 Data Block Example Using a Motorola S-Record Source File

In this example, the data blocks are named, respectively, S_Record_Address and S_Record_Data. S_Record_Address is assigned to a group of pins named Address1 and S_Record Data is assigned to a group of pins named Data1. The file statement specifies the formatted data record file from which to extract data and the data type to be extracted—either address or data.

At runtime, the test reads the data form the specified file statement in each data block. Thus, the first data block, S_Record_Address, reads 180 elements or bytes of address information from the formatted Motorola S-Record file "S_Record_Data" into the data block named S_Record_Address. In the second data block, S_Record_Data, the file statement reads 180 elements of data into the data block named S_Record_Data.

4.2.3 Data Block Example Using an Intel Hexademical Record

To download a single Intel Hex file into an Address data block and a separate Data data block, the VCL syntax differs only slightly:

```
data Hex_Record_Address to groups Addr
      file "Hex_Record_Data" 180 hex record address
end data


data Hex_Record_Data to groups Data
      file "Hex_Record_Data" 180 hex record data
end data
```

4.3 Formatted Records

Formatted data records are used to define the data for programming flash devices. The most common data records used for flash programming are the Intel® Hexadecimal and Motorola® S-Record formats. Inte®l Hex and Motorola® S-Record data records are formatted files. Data is extracted from the formatted file through the use of a data blocks.

4.4 Motorola S-Records

The Motorola S-Record file record format consists of five fields: the record type, record length, address, data, and the checksum byte for the record. This field order is shown in **Figure 4-2** below.

RT	RL	AAAA	DDDD	CC
Record Type (1 byte)	Record length (1 byte)	Address (2, 3, or 4 bytes)	Data	Checksum (1 byte)

Figure 4-2 Motorola S-Record General Record Format

RT — The record type is indicated by an S followed by a single character which defines whether it is a start record, data record or end record. The following record types are recognized by HP 3070 software: S0, S1, S2, S3, S7, S8, or S9. VCL reads only data records and ignores start and end records.

RL — Record length defines the byte count in the record. The byte count is the total number of bytes used by the address, data and checksum fields. Each byte is represented by two characters. HP 3070 software allows a maximum record length of up to 256 characters.

AAAA — 2, 3, and 4 byte addresses at which the data field is to be loaded into memory.

DDDD — From 0 to n bytes of executable code, memory loadable data, or descriptive information.

CC — The checksum is formed by taking the sum of all the bytes in the length address and data fields and then taking the one’s complement.

4.4.1 Record Types

The Motorola S-Record format record type is indicated by a capital S followed by a 1-character record type. HP 3070 software recognizes the following record types:

S0 = Start record.

S1 = Data Record (4 character address, 2 bytes, 16 bits).

S2 = Data Record (6 character address, 3 bytes, 24 bits).

S3 = Data Record (8 character address, 4 bytes, 32 bits).

S7 = End Record (used with 8 character addresses).

S8 = End Record (used with 6 character addresses).

S9 = End Record (used with 4 character addresses).

VCL reads only these record type fields and ignores the others.

NOTE

When programming flash devices with Motorola S-Record data:

- All values are hexadecimal
- Record types S0, S7, S8, and S9 are ignored
- Checksum values are ignored
- Unknown record types are ignored
- Lines not starting with an "S" raise an exception
- Duplicate addresses result in concatenated data

4.4.1.1 Start Record

Record type S0, the start record, is normally used to signal that other data records follow. It may be used to store additional information in the object file in the data field. The start record begins with the capital S start character ("S") followed a zero. The start record, S0, is followed by the 2-character byte count, a 4-character address ("0000") and a 2-character checksum. The following is an example of the simplest start record (spaces are included for clarity only and are not present in a real object file).

Example: `S0 03 0000 FC`

4.4.1.2 Data Record

The record types S1, S2, S3 are virtually identical and are the records which contain the actual data of the object file. The record begins with the start character ("S") followed by the 2-character byte count and the appropriately sized address field. The data bytes follow the address field and the record is terminated with the 2-character checksum. Below are examples of S1, S2, and S3 data record formats. Spaces are included for clarity only and are not present in the real object file.

Data Record Type Examples

```
S1 07 1FF0 1B2C3E4F 7F

S2 0B 002FF0 1B2C3E4F506172 8E

S3 15 FFFF0010 000102030405060708090A0B0C0D0E0F FF
```

4.4.1.3 End Record

The record types S7, S8 and S9 are also identical and are used to indicate the end of data if the address is 0 or the start address if the address is non-zero. The three different record types are required for the 3 different address field sizes. The record begins with the start character ("S") followed by the 2-character byte count and the appropriately sized address field. Generally, an end record has no data bytes. The record terminates with the 2-character checksum.

End Record Type Examples

```
S7 05 00000000 FA

S8 04 000000 FB

S9 03 0000 FC
```

4.4.2 Motorola S-Record Example

The following is an example of a Motorola S-Record file. It contains a start record, a 6-character address data record and the appropriate end record.

```
S0 03 0000FC

S2 08 1000F0 01020304 ED

S8 04 000000 FB
```

Which translates as follows:

Address:	Data:
1000F0	01
1000F1	02
1000F2	03
1000F3	04

4.4.3 Structure of a Motorola S-Record

Small example of Motorola s record:

```
S00B000044415441120492F4FF33333
S113000000FF000400040004000400040004000040004D1
S113001000040004000400040004000400040004BC
S113002000040004000400040004000400040004AC
S113003000040004000400040004000400040004 9C
S113004000040004000400040004000400040004 8C
S113005000040004000400040004000400040004 7C
S113006000040004000400040004000400040004 6C
S113007000040004000400040004000400040004 5C
S113008000040004000400040004000400040004 4C
S113009000040004000400040004000400040004 3C
S11300A000040004000400040004000400040004 2C
S11300B000040004000400040004000400040004 1C
S11300C000040004000400040004000400040004 0C
S11300D000040004000400040004000400040004FC
S11300E000040004000400040004000400040004EC
S11300F000040004000400040004000400040004DC
```

Start Record: `0B000044415441120492F4FF33333`

Record Type	Record Length	Address	Data	Checksum
S1	13	0000	00FF000400040004000400040004	D1
S1	13	0010	000400040004000400040004000400040004	BC
S1	13	0020	000400040004000400040004000400040004	AC
S1	13	0030	000400040004000400040004000400040004	9C
S1	13	0040	000400040004000400040004000400040004	8C
S1	13	0050	000400040004000400040004000400040004	7C
S1	13	0060	000400040004000400040004000400040004	6C
S1	13	0070	000400040004000400040004000400040004	5C
S1	13	0080	000400040004000400040004000400040004	4C
S1	13	0090	000400040004000400040004000400040004	3C
S1	13	00A0	000400040004000400040004000400040004	2C
S1	13	00B0	000400040004000400040004000400040004	1C
S1	13	00C0	000400040004000400040004000400040004	0C
S1	13	00D0	000400040004000400040004000400040004	FC
S1	13	00E0	000400040004000400040004000400040004	EC
S1	13	00F0	000400040004000400040004000400040004	DC

4.5 Intel Hexadecimal Records

The Intel 32-bit Hexadecimal Object file record format has a 9-character, 4 field prefix that defines the start of the record, byte count, load address, and record type. The record format also has a 2-character suffix containing a checksum.

An Intel Hex record is composed of five fields: load length, address, record type, data, and checksum. A colon defines the start of the record followed by a byte count (load length), address, record type, data, and checksum. Each character in the data file represents 4 bits of information.

This field order is shown below:

Figure 4-3 Intel Hex Record General Record Format

: LL	AAAA	RT	DDDD	CC
Figure 4-4 ":" Load Length	Address	Rec. Type	Data	Checksum

: A colon defines the start of the record.

LL Load length represents the number of data bytes in the record.

AAAA This is the 16 bit load address.

RT The Intel Hexadecimal Object file record format contains six record types.

00 = Data Record.

01 = End Record.

02 = Extended Segment Address Record.

03 = Start Segment Address Record.

04 = Extended Linear Address Record.

05 = Start Linear Address Record.

DDDD Data consists of from 0 to n bytes of executable code, memory loadable data, or descriptive information.

CC The checksum is formed by taking the sum of all the bytes in the length address and data fields, and then taking the one's complement.

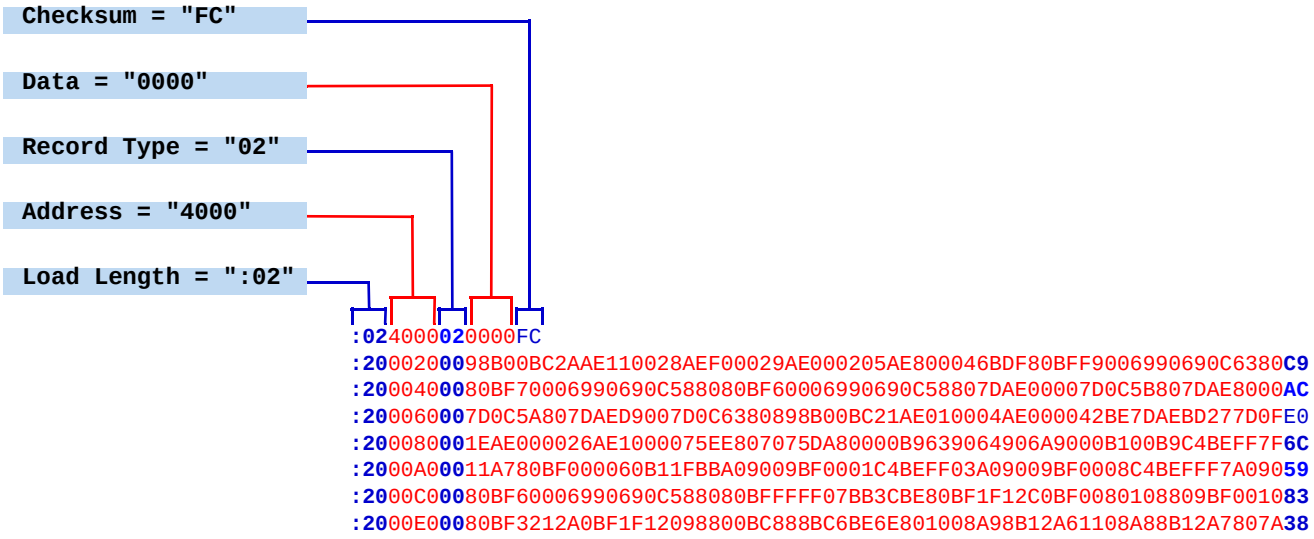


Figure 4-5 Intel Hex Data File

4.5.1 Record Types

The Intel 32-bit Hexadecimal Object file record format contains six record types

00 Data Record

01 End Record

02 Extended Segment Address Record

03 Start Segment Address Record

04 Extended Linear Address Record

05 Start Linear Address Record

4.5.1.1 Data Record

Record type 00, the data record, is the record which contains the data of the file. The data record begins with the colon start character (":") followed by the byte count, the address of the first byte and the record type ("00"). Following the record type are the data bytes. The checksum follows the data bytes and is the two's compliment of the preceding bytes in the record, excluding the start character. The following are examples of data records (spaces are included for clarity only and do not occur in a real object file).

```
:10 0000 00 FFFEFDFCFBFAF9F8F7F6F5F4F3F2F1F0 FF
:05 0010 00 0102030405 AA
```

4.5.1.2 End Record

Record type 01, the end record, signals the end of the data file. The end record starts with the colon start character (":") followed by the byte count ("00"), the address ("0000"), the record type ("01") and the checksum ("FF").

```
:00 0000 01 FF
```

4.5.1.3 Extended Segment Address Record

Record type 02, the extended segment address record, defines bits 4 through 19 of the segment base address. It can appear anywhere within the object file and it affects the absolute memory address of all subsequent data records in the file until it is changed. The extended segment address record starts with the colon start character (":") followed by the byte count ("02"), the address ("0000"), the record type ("02"), the 4 character ASCII representation of the hexadecimal number represented by bits 4 through 19 of the segment base address and the 2 character checksum.

```
:02 0000 02 1000 55
```

4.5.1.4 Start Segment Address Record

Record type 03, the start segment address record, defines bits 4 through 19 of the execution start segment base address for the object file. This record is currently ignored by the HP 3070.

```
:02 0000 03 0000 55
```

4.5.1.5 Extended Linear Address Record

Record type 04, the extended linear address record, defines bits 16 through 31 of the destination address. It can appear anywhere in the object file and it effects the absolute memory address of all subsequent data records in the file until it is changed. The extended linear address record starts with the colon start character (":") followed by the byte count ("02"), the address ("0000"), the record type ("04"),the 4 character ASCII representation of the hexadecimal number represented by bits 16 through 31 of the destination address and the 2 character checksum.

```
:02 0000 04 FFFF 55
```

4.5.1.6 Start Linear Address Record

Record type 05, the start linear address record, defines bits 16 through 31 of the execution start address for the object file.This record is currently ignored by the HP 3070 software.

```
:02 0000 05 0000 55
```

NOTE

When programming flash devices with Intel Hex data records:

- All values are hexadecimal
- Record types 03 and 05 are ignored
- Checksum values are ignored
- Unknown record types are ignored
- Lines not starting with a colon (":") raise an exception

4.5.1.7 Intel Hex Record Example

The following is an example of an Intel Hexadecimal Object file record. It contains the following records: extended linear address, extended segment address, data and end.

```
:020000040108EA
:0200000212FFBD
:0401000090FFAA5502
:00000001FF
```

1. Determine the extended linear address offset for the data record. (0108 in this example)

```
:02 0000 04 0108 EA
```

2. Determine the extended segment address for the data record.

(12FF in this example)

```
:02 0000 02 12FF BD
```

3. Determine the address offset for the data in the data record. (0100 in this example)

```
:04 0100 00 90FFAA55 02
```

4. Calculate the absolute address for the first byte of the data record

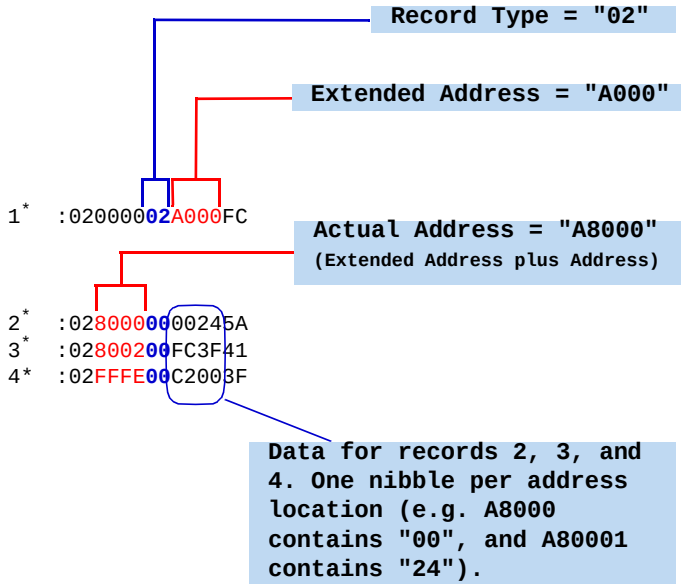
108 000	linear address offset shifted left 16 bits
+ **0001 2FF0**	segment address offset shifted left 4 bits
+ **0000 0100**	address offset from data record
= **0109 30F0**	32 bit address for first data byte

5. Which results in the following:

```
010930F0              90
010930F1              FF
010930F2              AA
010930F3              55
```

4.5.1.8 Extended Segment Record Example

The following Intel Hex example is comprised of two extended segment records and four data records. To better understand how extended addresses behave, focus on the addresses highlighted below:



* Data Number is not part of data record.

Figure 4-6 Extended Address Records

Though the addresses appear to be 8000, 8002, FFFE, 0000, and 0002, the compiler interprets these addresses as A8000, A8002, AFFFE. The example in **Figure 4-6** shows data records from 1 to 5. The following steps shows which addresses the data record will be programmed to...

1. The information following the "02" declaration offsets address subsequent addresses by "a000". This occurs on every address until another address record type occurs.

2. The data record immediately following the extended segment record (data record 2, in **Figure 4-6**, is added to "A000". The addresses and data for this record follows:.

```
  A000
+ 8000
------
  A8000  Data = 00
  A8001  Data = 24
```

3. Data record 3 of **Figure 4-6** will contain the following data at

the following addresses based on this calculation:

```
   A000
+  8002
   A8002  Data = FC
   A8003  Data = 3F
```

4. Data record 4 of **<HYPER T-11-ITAL>Figure -4-6 on page -14** will contain the following data at the following address based on this calculation:

```
   A000
+  FFFE
   AFFFE  Data = C2
   AFFFF  Data = 00
```

4.6 General Data Block Usage

Following is an example of a data block that shows some of the ways that data is defined in the block. The block begins with the "data" statement and ends with the "end data" statement. The "values" and "file" statements assign values to the block. In this example the block is named "Device_Data" and is assigned to a previously defined group of pins named "Data_Bus".

```
! in the Vector Definition section
     . . .
  data Device_Data to groups Data_Bus
     values 20
     values -104
     values 2, 384, -7, 66
     values A(3), A(9), A(21)
     values B(2:51)
     file "Data_File" 120
  end data
```

The example data block contains a total of 179 decimal values. In sequence, these are:

20, -104, 2, 384, -7, 66

three values from array "A"

fifty values from array "B"

one hundred and twenty values from a file named "Data_File"

Here are some general notes about the values shown in the example.

- Each value is saved as a 32-bit, twos-complement, number. The values are binary and are not affected by the format that is in effect. A value is applied to a pin group in the same way as an array element is applied. For example, with a 3-pin group, the value 6 applies bits 1 1 0 to the pins in the order the pins were assigned to the group. Where fewer than 32 bits are required, the higher-order bits are ignored.

- Arrays in VCL are zero-based. Therefore, the values from array "A" are from its fourth, tenth and twenty-second elements. "B(2:51)" indicates that 50 consecutive values are to be taken from array "B", from elements 2 through 51, inclusive. These arrays are passed to the test from the testplan.

- If the file named in the "file" statement is not in the same directory as the test, the file id must indicate the file's path name. If the file contains values, they must be integers, with only one value on each line; if the file contains formatted records (described later), each line must contain only one such record. In the example shown above, 120 consecutive numbers will be read from the file at run time. Reading always starts at the beginning of the file.

- The file should contain at least the number of values specified in the "file" statement. If there are more, the extra values will be ignored; if there are fewer, "1" states will be driven at run time for the missing values. The receivers for missing received values will be set to their high-impedance states.

Following is an example of vector execution statements that read the data from block "Device_Data" and apply it to the assigned group of pins, "Data_Bus" from the preceding example. The vector execution statements use their "drive data" and "receive data" parameters to access the data block.

```
! in the Vector Execution section
        . . .
     repeat 179 times
        homingloop 25 times
              . . .
           execute Write
           execute Keep_Control drive data Device_Data
           execute End_Cycle
              . . .
           execute Verify
           execute Keep_Control receive data Device_Data  exit
if pass
           execute End_Cycle
              . . .
        end homingloop
        next Device_Data
     end repeat
        . . .
     rewind Device_Data
        . . .
```

The preceding example is a simplified version of a loop intended to program and verify flash programming. In the homingloop, the first value from the data block, "Device_Data", is written into the flash RAM and then verified. The exit occurs from the homingloop when the data is read correctly. Since this is a true counted homingloop, the test stops after the 25th iteration of the loop if the correct data is not read.

After the exit from the homingloop, the "next" statement increments the data counter by one, so that, the next time through the homingloop, the next value will be used from the data block. Notice that, until the "next" statement is executed, all "drive data" and "receive data" parameters reference the same element from the data block. At the end of the repeat loop the "rewind" statement enables you to reuse the data in the block. It resets the counter in the data block so that the next loop can start reading data at the beginning of the block.

As with array elements, the vector that drives or receives an element from the block also executes any other states that are specified for that vector in the Vector Definition section of the test. Data blocks cannot be specified in a PCF vector.

A "drive data" must apply to a group that consists either of input pins or of bidirectional pins, and a "receive data" must apply to a group that consists of output pins or bidirectional pins. The direction of a group of bidirectional pins is determined by the "drive data" or "receive data" parameter in the vector execution statement. If there is a conflict, those parameters override the direction specified in the vector definition. The parameters also override any conflicting states set on any group of pins in the vector definition.

Any one vector can drive or receive elements from more than one data block. The element from any one block can be applied only to the pin group named in the definition of that block. Any one pin group can be associated with only one data block. Since the "next" statement names the data block to which it applies, it allows you to increment the data counters for the blocks independently.

Other rules that apply to the use of data blocks are:

- Data blocks cannot be used on the pins in the count fields in counters.
- Pins that are be used with a data block can be assigned to only one group. If they are assigned to more than one group, only the last assignment will be valid.
- An error occurs if a group that is assigned a data block contains an asterisk.

3.4.1 Testing Single-Byte Devices with Data Records

The data block can also be used with the formatted Flash RAM records. Either Intel® Hex Format or Motorola® S-Records can be used. These records have an address part and a data part. The file name is specified in the data block definition, as shown in the example below. Single-byte programming is used when no more than eight bits are to be programmed at a time.

This example creates two data blocks that use formatted data. The first block reads only the addresses from the file. These are the addresses where the data belongs in the device under test. The second block reads only the data to be written to, or read from, those addresses. The data type ("address" or "data") is specified in the "file" statement in each block. At runtime, the test reads the data from the specified file and selects either the addresses or the data from each record, as specified by the last parameter in the "file" statement. In this example, 128 addresses and 128 bytes of data, are read from the file.

```
data S_Record_Address to groups Address1
     file "S_records" 128 s record address ! selects
"address" part of each record
end data

data S_Record_Data to groups Data1
     file "S_records" 128 s record data  ! selects "data" part
of each record
end data
```

Blocks for the Intel® Hex Format would be similar except that keyword "hex" is substituted for keyword "s" in the "file" statement. For example:

```
file "Hex_Format_Data" 128 hex record address
```

These data blocks are referenced in the vectors in exactly the same way as the data blocks that contain numeric data, described earlier. Following is an example of a loop that uses the two data blocks defined above. The example assumes that pins 8 through 1, in that order, were assigned to group "Data1":

```
repeat 128 times
     execute Read drive data S_Record_Address receive data
S_Record_Data
     next S_Record_Address
     next S_Record_Data
end repeat
```

On each pass through the loop, eight bits (one byte) would be read from group Data1, with the least significant bit read from pin 1.

3.4.2 Testing multibyte Devices with Data Records

Data blocks can also be used with multibyte-wide devices, enabling you to test memory devices, and banks of devices in parallel, up to whatever width your records will allow. The number of bytes is determined by the number of pins assigned to the data bus — there will always be a whole number of bytes. For example, 16 pins results in a width of two bytes, and 28 pins results in a width of four bytes; in the latter case, the four most significant bits in the fourth byte of data from the record would be ignored. Following is an example of a program that reads data from a device two bytes at a time.

```
assign Data_Bus to pins 16, 15, 14, 13, 12, 11
assign Data_Bus to pins 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
     . . .
data S_Record_Address to groups Address_1
     file "S_records" 128 s record address
end data

data S_Record_Data to groups Data_Bus
     file "S_records" 128 s record data
end data
unit
     repeat 64 times
        execute Read drive data S_Record_Address receive data
S_Record_Data
        next S_Record_Address
        next S_Record_Data
     end repeat
   end unit
```

Because there are more than 8 pins, VCL automatically assumes multibyte programming, in this case two bytes wide. In the Vector Execution section, each time the data is accessed, two bytes will be read, which is why the loop is repeated only 64 times. The "next" statements increment the data and address counters by two (that is, by the byte width).

The bytes are assigned to the pins in reverse order, with the least significant bit assigned to the pin on the right. In this case, the first byte read is assigned to pins 8 through 1, with the most significant bit on pin 8. Similarly, the second byte is assigned to pins 16 through 9. Notice that you can change the order that the bits will be assigned to the pins, simply by rearranging the pin order in the "assign" statements.

If there were, say, only 12 pins (12 through 1), then the four higher order bits in the second byte would be discarded. Starting from the right, bits 0 through 3 would be assigned to pins 9 through 12, respectively, and bits 4 through 7 would be ignored.

Addressing in the file must be arranged so that, in each group (byte width) of bytes that is read from the file, the first byte must have an appropriate address. That is, an address whose value is divisible by the byte width with no remainder — 0, or 2, or 4, or 6... in the case of 2-byte width. Also, the addresses of the bytes in each group must be contiguous. Addresses with up to 32 bits can be accommodated.

With multibyte programming, VCL automatically sets the byte-width to fit the number of pins in the data bus, as described above. It also increments the addresses that are generated by the same value as the byte-width. If four bytes are to be read, the addressing will be incremented by four. In this case, the address of the first byte in each group of bytes must be a value that is divisible by four with no remainder (0, 4, 8...).

The "file" statement has an optional "step" parameter that can be used in the special case where you are writing to, or reading from, multiple devices in parallel but addressing them individually. As an example, suppose you have two one-byte wide devices. In this case, your data bus is two bytes wide but you need to increment the address of each device only by one. Assuming a data bus with 16 pins, the following statement will allow the address that is driven by the test to be incremented by one for each two bytes that are read in parallel:

```
file "Hex_records" 128 hex record address step 1
```

You can have any step value provided that it is consistent with the structure of your circuit and of your data records. The byte-width must be evenly divisible by the step value. The actual addresses that are driven to the address bus are calculated as

(logical address * step value) / byte width

For each set of data read, the "logical address" is the address in the record of the first byte.

For the above example, where the data is two bytes wide, if the logical addresses of the data are 0, 2, 4..., the driven addresses are 0, 1, 2 . . . .

Chapter 5
VCL Syntax for Flash OBP

Rev. A

This chapter describes:

5.1 Overview

Programming flash devices requires an understanding of Vector Control Language (VCL), the structured programming language used to write flash digital tests. Flash70 introduces several new VCL statements that allow you to control and modify flash digital tests. Read this chapter to learn about the structure and usage of VCL syntax statements for flash programming.

5.2 VCL Syntax in Flash Digital Tests

Vector Control Language (VCL) is a structured programming language used to write tests for individual digital devices or clusters of digital devices. A VCL test applies a series of vectors to the device and compares the device's actual responses to the expected responses. Based on these responses, a test either passes or fails.

A flash VCL test is composed of VCL programming statements. VCL statements are organized into test sections that perform specific functions when the test is executed. This section describes the VCL syntax conventions for flash programming.

NOTE: The "program" test examples provided in this section are written for an Intel 28f160 flash device and use a Motorola S-Record data file named "2Mx16.data"

5.3 The Structure of a VCL Test

A VCL test contains four sections, each composed of a set of programming statements. These sections must appear in the following sequence: 1.) Declaration, 2.) Timing[1] , 3.) Vector Definition, and 4.) Vector Execution.

5.3.1 Declaration section

The Declaration section defines the type of test, device characteristics such as inputs and outputs, and the voltage levels required to the test the device.

5.3.2 Timing section

The Timing section defines the timing sets. Timing sets control the formatted drivers and receivers and the vector drive and receive times. Timing sets cannot be used with the Flash70algorithm. Refer to the HP 3070 Users' Manual for more information. See "System Clocks & Test Timing" and "Timing Sets" in **Test Methods: Digital, 3.3 & 3.4.**

1. If a vector cycle/receive delay is defined, the timing section should appear first in a VCL test.

5.3.3 Vector Definition section

The Vector Definition section defines a series of vectors, each of which contains a pattern of bits to drive to the device and the pattern of bits expected to be received in response from the device.

5.3.4 Vector Execution section

The Vector Execution section contains a series of vector execution statements that apply the vectors in the required order to determine whether or not the device is operating properly.

5.4 Placing Flash VCL Statements in a Test

VCL statements must appear in the appropriate section of a flash test. The sections of a VCL test and the syntax for implementing flash tests are described below.

5.4.1 VCL Statements in the Declaration Section of a Test

The following VCL statements belong in the declaration section of a digital flash test. These statements are likely to appear in most flash tests.

```
flash
generate static test
family
dynamic
```

5.4.1.1 Example Declaration Section for a Flash Test

The following example demonstrates the VCL statements and syntax commonly used in the Declaration Section of a flash test.

```
!!!!   >  >  >   Declaration Section  <  <  <    !!!!

    flash
    generate static test
    assign Address_bus to pins 4, 5, 6, . . .
32
    assign Data_bus to pins 52, 50, 47, . . .
33

    family Flash_5V
    dynamic Data_bus, Address_bus
```

Figure 5-1 Flash VCL Declaration Statements

These **Flash VCL Declaration Statements** have the following effect:

`flash`

The "flash" statement improves data utilization and flash programming speed.

`generate static test`

The optional "generate static test" statement instructs the compiler to incorporate the data file directly into the object file of the "program" test.

`dynamic Data_bus, Address_bus`

The "dynamic" statement assigns dynamic resources to all pins listed in the "Data_bus" and "Address_bus" assignment statements. This statement is not required with Flash70.

5.4.1.2 .Description of Declaration Statements

5.4.1.2.1 flash

The "flash" statement activates the compiler features that improve data utilization and programming speed. If Flash70 software is installed on your test system, you can use this statement to enhance programming speed. Enter this statement at the beginning of the test in the Declaration section. Also, if you wish to use the faster Flash70 algorithm, enter the statement "enable Flash70" in the board "config" file.

NOTE 

Flash inhibit statements allow you to selectively use Flash70 features. Inhibit statements are inserted into the test in the Declaration section. See **Syntax to Inhibit Flash70 Algorithm, on page 5-13**.

5.4.1.2.2 generate static test

The "generate static test" VCL statement speeds up the first run times of test files that use segmented data (e.g. the "program" and "verify" flash OBP tests). This statement instructs the compiler to include the data file contents in the test object file (e.g. "u8:program.o"). If the data file changes on a regular basis, this statement is not typically used, since data changes require recompilation.

The "generate static test" statement is used only in executable tests and is ignored during a library compilation. If the data files are not available when the executable test is compiled, an error occurs. If an error occurs, supply the required data files or comment out the "generate static test" statement in the "program" or "verify" test before recompiling.

The "generate static test" statement must appear at the beginning of digital test files before pin assignment declarations. When using this statement, speed improvements depend on the size of the data file being programmed.

NOTE 

Although the "generate static test" statement improves first run times significantly, the object file that is created can be very large. If disk space is limited, review the size of the object files precompiled with "generate static test".

5.4.1.2.3 family

The "family" VCL statement declares the logic family of the device under test. The "family" statement enables the compiler to generate the setup code for the drivers and receivers on the pin cards in the testhead. Any one of the following "family" statements can be included in the Declaration section of the VCL test: "`Flash_5v`", "`Flash_3v`", "`Flash_2.2v`", "`Flash 1.8v`".

NOTE 

Multiple flash families can be declared in the "board" file. However, only one flash family should be called by the digital libraries of your flash test suite.

5.4.1.2.4 dynamic

If you are not using Flash70 software, this statement dramatically improves the speed of flash device programming. The "dynamic" VCL statement appears in the Declaration section of a VCL test to instruct the Module Pin Assignment software to assign dynamic resources to a specified groups of pins. Since assigning pins to dynamic resources has first priority, the "dynamic" statement needs to appear in one test, generally, the "program" test. This statement must be included during initial program development to ensure that adequate resources are assigned.

NOTE 

Although "dynamic" pin assignments do not improve Flash70 programming speed, this statement should be used for backward compatibility.

5.4.2 Flash VCL Statements in the Definition Section of a Test

In the definition section of a flash test, the data file used for programming is specified by the "file" statement. Also, vectors are defined in the definition section. In the **Flash VCL Definition Statements, on page 5-7**, the statements that appear as blue text provide examples of flash VCL statements included in the definition section.

5.4.2.1 Example Definition Section of a Flash Test

```
!!!!   > > >   Definition Section   < < <   !!!!
vector Initialize
.
.
.

vector Keep_Control
.
.
.
vector Three_state
.
.
.
vector XSR_Ready
.
.
.

vector Data_W
   initialize  to Keep_Control
   drive data_bus ! bidirectional group of pins declared as inputs
   set  Address_bus to "000000"
   set  WE_bar  to "0"
   set  Data_bus  to  "0000"
end vector
.
.
.

data Data to groups Data_bus
   file "2Mx16.data" 1048576 s record data
end data

data Address to groups Address_bus
   file "2Mx16.data" 1048576 s record address
end data
```

Called by execution statement in Figure 5-4.

These two data blocks define the variables used by vector "Data_W" and the "execute. . . drive" statement in Figure 5-4.

Figure 5-2 Flash VCL Definition Statements

The file statements shown in Figure 5-2, the **Flash VCL Definition Statements** have the following effects:

- The "file" statements enable the flash digital tests to program 1,048,576 bytes of formatted data from a Motorola S Record named "2Mx16.data".
- Both address and data information is contained within the file "2Mx16.data".
- In the execution section of the test, the vector "Data_W" is used as a template to drive the address and data from the data record to the "Data_bus" and "Address_bus" pin groups.

5.4.2.2 Description of Definition Statements

5.4.2.2.1 file

The "file" statement is used to assign values from a data source file to a data block. This statement appears in digital tests that use dynamic data (i.e. the "program" and "verify" flash OBP tests). When used for flash programming, the "file" statement points to a formatted data file–usually a Motorola S-Record or Intel Hex record. These records are parsed into an address and a data section when they are read by a flash data block. Although the file statements in Figure 5-4, the **Vector Declaration and Execution** section, refer to a single data file ("2Mx16.data"), you can use multiple formatted source files to program address and data information.

5.4.2.2.2 file statement option

For detailed information on data blocks and the various uses of the file statement, see **File Statement Options, on page 5-14**.

5.4.3 Flash VCL Statements in the Execution Section of a Test

Programming flash devices with Motorola S-Records or Intel Hex involves the same execution pattern: a repeat loop that programs segments of a data file onto a flash device. The execution section of a flash "program" or "verify" test is likely to contain the following VCL statements:

```
segment
repeat [number] times
execute
drive
next address
next data
```

Examples of these syntax statements appear in Figure 5-3, **Flash VCL Execution Statements**.:

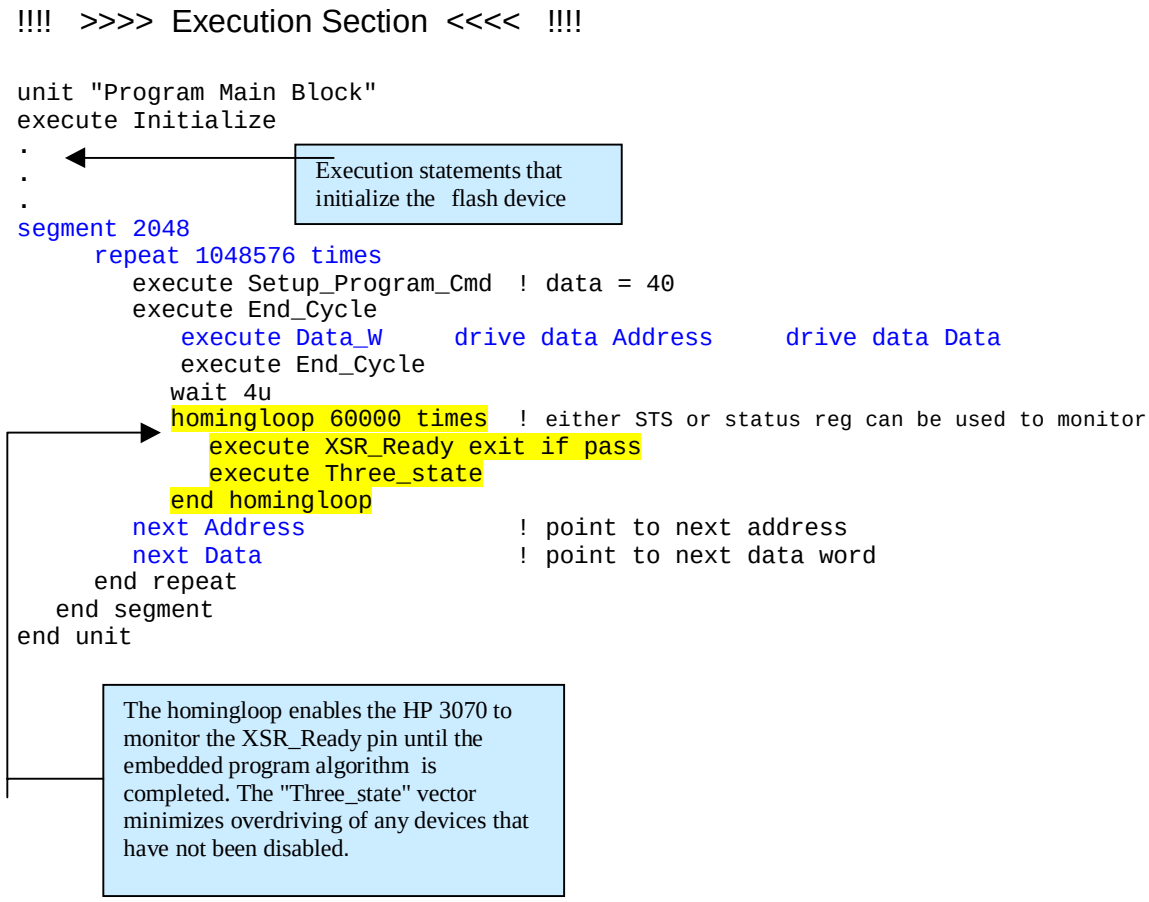


Figure 5-3 Flash VCL Execution Statements

From the execution statements shown in **Figure 5-3,** you can determine the following information about the test:

The "segment" statement indicates that the Motorola S-Record data from "2Mx16.data" file (shown in the definition section, **Figure 5-2**) will be programmed onto the flash device in 2048 word[1] increments until the 1,048,576 word repeat loop is complete.

1. For our 28f160 example, one word is 16 bits of information.

- The amount of data programmed or verified by the test is defined by the "repeat *[number]* times" statement. Counted by words of data, the number usually matches the size of the flash device, or if the test is optimized for performance, the Mb size of the data file.

- The "drive" statement applies the data from the file defined in the data block statements shown in Figure 5-2 (the file is defined as "2Mx16.data"). Address and Data variables are passed to the "Data_W" vector to drive the address bus and data bus of the flash device.

- The "next Address" and "next Data" statements increment the counter to the next address and next word of data to be programmed in the segment.

5.4.3.1 Description of Flash VCL Execution Statements

5.4.3.1.1 segment

HP 3070 pin cards do not have enough memory to program an entire flash device at once. The "segment" VCL statement allows portions of the data file to be programmed sequentially.

The segmentation value represents the number of times these vectors will be saved in the vector RAM during one download. You cannot calculate this number because you do not know how much RAM space will be available. The compiler calculates the space required to save the compressed vectors during a programming operation. You can determine a suitable number for segment size by using the *guess and compile* technique described in the HP 3070 Users' Manual, Test Methods: Digital.

Flash70 segment size

If you are not using Flash70 software, segment size is determined by the vector RAM behind the pins, which can never exceed 8k. The Flash70 algorithm enables use of sequence RAM for determining segment size. With the ControlXT cards, this means segments can be up to 1 MB in size. This use of memory greatly improves programming time.

For more information on Flash70 and ControlXT cards, see **Chapter 8, "Series and Parallel Programming."**

5.4.3.1.2 repeat

The repeat loop represents the number of words of data that will be either programmed or verified by a "program" and "verify" test. The size of the repeat loop usually corresponds to the total flash device size or the total data file size. Once the first segment of data is programmed onto the flash device, the segment ends with the "end segment" statement, and then the HP 3070 makes another pass through the "program" test. The VCL code in the Execution Section in **Figure 5-3** instructs the compiler to make 512 passes through the test. This number is derived from the size of the data, 1,048,576 words, divided by the segment size, 2048 bits.

For example, the first pass through the "program" test drives words 1 through 2048, the second pass programs 2049 through 4096, and the third pass programs 4097 through 6144, etc. This process continues until 1,048,576 words of data are programmed (2 **Mb** for our 28f160 example).

For more information on the repeat loop syntax, click on the Syntax Reference button at the bottom of the page and search for "repeat."

5.4.3.1.3 execute and drive

In flash OBP digital tests, the "drive" statement appears in the Vector Execution section of the test. The "drive" statement parses addresses and data from the formatted data file. It applies this data to a vector, and this vector drives the data pin states to the device being programming. The "execute" statement shown in **Figure 5-4** calls the "Data_W" vector, parses the address and data records to the variables "Address" and "Data", and then the vector drives the pins belonging to the "Data_bus" group. In this manner, every pass through the repeat loop programs one word of the file "2Mx16.data" onto the flash device.

The relationship between the programming statements in the Vector Definition and Vector Execution sections is shown in **Figure 5-4.**

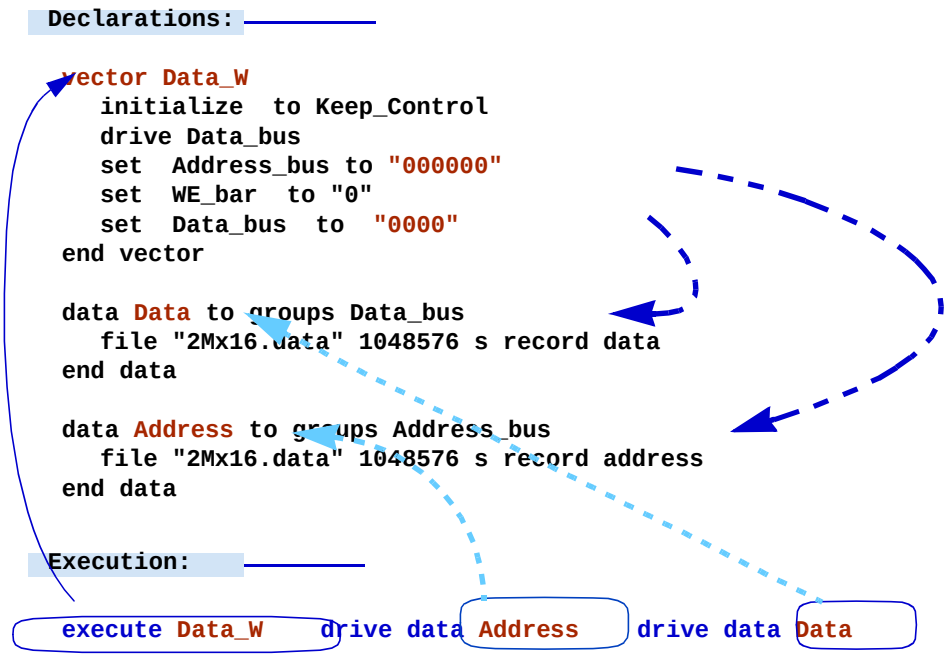


Figure 5-4 Vector Declaration and Execution

Notice that the "drive data Address" and "drive data Data" statements reference the data block definitions for "Address_bus" and "Data_bus". This reference determines which data record in the file should be programmed on this pass through the repeat loop. The "set" statements in vector "Data_W" is a place holder for the "drive data Data" and "drive data Address" values.

5.4.3.1.4 next

The "next" statement increments the data records by reading the next value and calculating data length and address fields of the specified data file. In the 28f160 example, the "next Address" and "next Data" statements instruct the HP 3070 software to proceed to the next data value in "2Mx16.data".

5.5 Syntax to Inhibit Flash70 Algorithm

In some cases, you may not wish to update existing tests using version B.03.00 or later software. VCL provides syntax to turn off some or all of the new flash features. To selectively turn off features, add one or more of the inhibit commands to the beginning of a flash test (usually, "program" or "verify" tests) before any "assign to" statements. The inhibit flash commands follow:

5.5.1 Turning off the Flash70 Algorithm

The syntax to turn off the Flash70 algorithm is:

```
generate flash inhibit dynamic sequence
```

5.5.2 Turning off All Flash Features

The following statement removes all B.03.00 flash features from a given OBP "program" or "verify" test, creating a B.02.75 version of the library.

This statement must appear in the Declaration section of a test before "assign to" statements:

```
generate flash inhibit all
```

5.5.3 Turning off Limited Addressing

The new flash software will not write outside the address size range of the flash device being programmed. Enter the following statement into the "program" or "verify" test to disable this feature.

This statement must appear in the Declaration section of a test before "assign to" statements:

```
generate flash inhibit limited address
```

5.5.4 Turning off Segment Removal

The new flash software does not continue generating segments beyond the end of the data file. If you want the "program" or "verify" test to program the entire flash part without regard to the

data file size, enter the following statement to the "program" and "verify" test.

This statement must appear in the Declaration section of a test before "assign to" statements:

```
generate flash inhibit segment removal
```

5.5.5 Turning off Data Removal

Flash70 software strips the FFH sequences from Intel Hex and Motorola S-Record data files. Enter the following statement into the "program" and "verify" tests to disable this feature.

This statement must appear in the Declaration section of a test before "assign to" statements:

```
generate flash inhibit data removal
```

5.6 File Statement Options

Flash tests program flash memories in units of data called segments. The size of the segment defined in the VCL test determines how many cells must be programmed at one time. The HP 3070 cannot stop programming a segment in the middle of execution, even if the data to be programmed has been exhausted. Since each segment must be programmed in its entirety, some valid form of data must be programmed into the remaining cells if the data runs out in the middle of a segment.

HP 3070 software version B.03.00 and later provides several options for handling the end of data condition in a flash test using file data. You can choose one of the following options:

- Default

  Current method: All drivers drive to high state, All receivers don't care.

- Reuse FF

  Program locations currently programmed with FF data with FF data.

- Unused FF

  Program locations in the address range but not in the data file with FF data.

- User FF

  User provides an address and FF data is programmed at that address.

NOTE

The B.03.00 end of data options are available only on HP 3070 Series II or later. Also, since some flash devices can be damaged by continuous programming of data other than FFFFFFFFH, none of these options allow it.

5.6.1 Default

The default method for handling the end of data condition is to program all dynamic drivers to the high state and all dynamic receivers to the don't care state when data is no longer available. All existing flash tests, all tests targeting the HP3070 Series II, and all tests encountering the end of data condition when using variable data or ASCII files use the default method.

The default method is predictable. The end of data condition always results in attempts to program location FFFFFFFFH with FFFFFFFFH. Problems typically occur when location FFFFFFFFH has already been programmed to something other than FFFFFFFFH.

The file statement without an end of data modifier implies the use of the default method. A end of data default modifier to the file statement is included for use with the "reuse", "unused" and "user" methods.

The following syntax results in the default end of data handling method:

```
file "data" 131072 s record data
file "data" 131072 s record data default
```

5.6.2 "reuse" Data Modifier

The reuse method of handling the end of data condition utilizes locations in the data file containing FFH. These locations provide data to the dynamic drivers and dynamic receivers on end of data. This feature will work with the FFH data removal feature. The system removes FFH data from the normal data stream but still remembers these locations for use in end of data handling.

The reuse method always programs locations with the same data. There is no danger when programming location FFFFFFFFH unless it has previously been programmed with FFFFFFFFH. Problems result when the user programs these FFH locations prior to programming the device using the record file. Serial numbers, dates and other small pieces of information may be programmed in these areas prior to programming the rest of the device data. Swapping the order of these programming cycles will remedy this problem.

A more insidious problem lies in record files containing no FFH data locations. It is impossible to reuse an FFH location that doesn't exist (as far as the record file is concerned). One possible solution is to adjust the repeat loop and segment size to match the data file size.

The file statement modifier to support the reuse method on the end of data condition is "reuse" as shown below:

```
file "data" 131072 s record data  reuse
```

5.6.3 "unused" Data Modifier

The unused method of handing the end of data condition uses locations not contained in the record file subject to the constraints placed on the address space. The locations are programmed to FFFFFFFFH on end of data. This method will not use FFH locations removed by the skip data feature.

This method requires locations not in the record file to be erased. Serial numbers, dates, and other unique identifiers for a board may not have locations entered in the record file but may be programmed prior to programming with the record file. This situation can cause failures during programming with the record file.

The file statement modifier to support the unused method on the end of data condition is "unused" as shown below:

```
file "data" 131072 s record data  unused
```

5.6.4 "user" Data Modifier

The user end of data method allows you to specify the location at which to program FFH data. The location specified is relative to the address space of the record file and not to the **DUT**.

Implementing the user method can cause two problems. First, the specified user address will probably change with each data file revision. Second, the specified address may be programmed with data other than FFH in the record file.

The file statement modifier to support the user method on the end of data condition is "user *[integer address]*" as shown below.

```
file "data" 131072 s record data  user 0
```

5.7 28f160 "u8:program" VCL Example

Since it is sometimes easier to understand VCL statements within the context of a complete test, the key elements of the "u8:program" test for an Intel 28f160 part follows:

NOTE

The following excerpt includes sections of the test that illustrate the syntax provided in this chapter. To view an entire test, choose any library listed under "/hp3070/library/supplemental/flash".

```
!------------------  Declaration Section  ----------------!
flash
assign   Address_bus to pins   4, 5, 6, 7, 8, 10, 11, 12
assign   Address_bus to pins   13, 17, 18, 19, 20, 22, 23, 24
assign   Address_bus to pins   25, 26, 27, 28, 32

assign   Data_bus    to pins   52,50,47,45,41,39,36,34
assign   Data_bus    to pins   51,49,46,44,40,38,35,33
assign   Data_Status    to pins   51,49,46,44,40,38,35,33

! ---  END of Pin config for TSOP package ---  !

family   FLASH_5V

dynamic  Data_bus, Address_bus

!------------------  Definition Section  ----------------!
vector  Initialize
   drive Data_bus
   set   CE_bar      to "00"
   set   OE_bar      to "1"
   set   WE_bar      to "1"
   set   WP_bar      to "1"
   set   RP_bar      to "x"
   set   Byte_bar    to "x"
   set   Address_bus to "000000"
   set   Data_bus    to "0000"
end vector

vector  Keep_Control
   drive Data_bus
   set   CE_bar      to "kk"
   set   OE_bar      to "k"
   set   WE_bar      to "k"
   set   WP_bar      to "k"
   set   RP_bar      to "x"
   set   Byte_bar    to "x"
   set   Address_bus to "kkkkkk"
   set   Data_bus    to "kkkk"
end vector

vector  Three_state
   receive Data_bus
   set   CE_bar      to "zz"
```

```
   set   OE_bar      to "z"
   set   WE_bar      to "z"
   set   WP_bar      to "z"
   set   RP_bar      to "z"
   set   Byte_bar    to "z"
   set   Address_bus to "zzzzzz"
   set   Data_bus    to "zzzz"
end vector

vector  XSR_Ready
   initialize        to Keep_Control
   receive  Data_bus
   set  Address_bus  to "zzzzzz"
format binary Data_Bus
   set   Data_bus    to "XXXX XXXX 1XXX XXXX"
format hexadecimal Data_Bus
   set   OE_bar      to "0"
end vector

vector  Data_W
   initialize        to Keep_Control
   drive Data_bus
   set   WE_bar      to "0"
   set   Data_bus    to "0000"
end vector

data  Data     to groups   Data_bus
   file  "2Mx8.data" 1048576 s record data
end data

data  Address  to groups   Address_bus
   file  "2Mx8.data" 1048576 s record address
end data

!------------  Execution Section  -------------!

unit "Program Main Block"
  execute Initialize
   execute Clear_Sts_cmd              !!Command = 50H
   execute End_Cycle
   execute Status_Read_cmd            !!Command = 70H
   execute End_Cycle
   execute OE_true
   execute XSR_Ready
   execute End_Cycle

   segment 2048
      repeat 1048576 times
       execute Setup_Program_Cmd    ! data = 40
       execute End_Cycle

       execute Data_W   drive data Address     drive data Data
       execute End_Cycle

       wait 4u
         homingloop  60000 times
!!        either STS or status reg  can be used to monitor
          execute XSR_Ready exit if pass
          execute Three_state
```

```
        end homingloop
      next Address                !point to next address
      next Data                   !point to next  data word
     end repeat
    end segment

end unit

unit "Check Status"
    execute Status_Read_cmd          !!Command = 70H
    execute Write
    execute End_Cycle
    execute Check_Status_1xx0000x    !!7654 3210 (Status Reg.)
end unit                          !! |||| ||||
                                  !! |||| |||*- Reserved
                                      !! |||| ||*- Device protect status
                                      !! |||| |*- Program suspend status
                                      !! |||| *- Vpp Status
                                      !! |||*- Program Status
                                      !! ||*- Erase Status
                                      !! |*- Erase Suspend Status
                                      !! *- Write State Machine Status
```

Figure 5-5 Complete 28f160 "u8:program" VCL Test

Chapter 6
Generating Flash Digital Tests

Rev. A

This chapter describes the tasks and procedures for generating flash digital tests, including:

- The tasks required to develop flash digital tests. See **Flash Test Development Tasks, on page 6-3**.
- The flash OBP test development environment, including flash test descriptions and OBP task flow. See **Section One: Flash OBP Programming Steps, on page 6-5**.
- The procedure for developing flash tests with B.03.00 software. **Section Two: Steps to Developing Flash Digital Tests, on page 6-13**.
- The procedure for utilizing existing digital test libraries to develop flash tests. See **Section Three: Flash Tests and Existing Fixtures, on page 6-24**.

6.1 Overview

HP 3070 systems have been capable of on-board programming for some time. However, now the HP 3070 Series 3 and Flash70 software features make it easier and faster to implement on-board programming.

Flash device library models automate many flash in-circuit programming tasks, when used with the HP 3070 Series 3 and Flash70 software. These flash device library models serve as templates and can be used to program the most common flash RAM devices. If a particular flash device is not included in the standard library, you can search for a flash library test suite that is similar to your custom device and modify the test as needed.

The HP 3070's automated test generation software, IPG, uses HP 3070 standard libraries or custom digital library models to generate a suite of VCL tests for programming flash devices. These VCL tests contain programming statements that trigger embedded algorithms within a flash device. Embedded algorithms perform functions such as flash device erasure and manufacturer device identification. An IPG generated digital library test called "program" actually programs the flash device using formatted data. This formatted data source is typically from an external Intel® Hex or Motorola® S-Record file. IPG creates six **OBP** digital tests for each type of flash device installed on a given board. These tests are used in combination to develop a test suite for production on-board programming of flash memory devices.

6.1 Flash Test Development Tasks

Flash test development involves a series of tasks. To develop flash digital tests, perform the following tasks:

Add the appropriate enable commands to the board configuration file. For example, to enable Flash70 features in your board configuration file, add the statement **"enable flash70"** to the "config" file in the board directory.

For more information, see **Section Two: Steps to Developing Flash Digital Tests, on page 6-13**.

1. Verify that the local "board_defaults" file lists your flash device families under the FAMILY OPTIONS section.

   - Locate the "board_defaults" file in the directory: "**hp3070/standard/ "**

   - If your flash device families are not listed in the Family Options section of the "board_defaults" file, modify the file to include them. Enter the new flash family name in the Family Options section of the standard "board_defaults" file.

     NOTE: The "board_defaults" file provides values for HP Board Consultant to use in the "board" file when those values are not supplied by data files or by data entry.

2. Verify that the LIBRARY OPTIONS section of the "board_defaults" file contains the statement: **"/hp3070/library/supplemental/flash"**. Add the statement if it is missing. Its purpose is to point to the directory that contains the new B.03.00 flash library tests.

3. Ensure that the device part number specified in the "board" file matches the appropriate PDL file name found in the HP 3070 libraries. If they don’t match, change the name appropriately.

   Updated PDL files list flash digital tests in a format that references the library directory path. This directory path points to the name of the flash library device model, as shown in the following example:

   "/hp3070/library/supplemental/flash/am29f040_blank"

4. Run HP IPG Test Consultant to generate tests.

5. Modify the "testplan" file to include flash tests that reside in **"/hp3070/boards/board_files_dir/digital"**.

   For more information, see **Step 4: Modifying the 'testplan', on page 6-21**.

6. Verify that disabling is properly implemented in the IPG created flash tests.

   For more information, see **Step 3: Running HP Test Consultant, on page 6-17**.

7. If you are using combinational testing ("enable combo" appears in your config file) or Flash70 software, locate the digital directory of your application and comment the homingloop waits in your VCL "program" and "verify" test. Also, uncomment the hardware wait commands.

   For more information, see **Step 3: Running HP Test Consultant, on page 6-17**.

8. Rerun IPG Test Consultant to update tests with new disable signals and hardware triggers in the fixture.

6.2 Section One: Flash OBP Programming Steps

In the HP 3070 in-circuit test environment, six steps can be incorporated into the flash memory programming process: *id*, *blank*, *erase*, *verify*, *program*, and *crc*. These flash programming steps have a one-to-one correlation with the six IPG generated digital tests. With the exception of *erase* and *program*, the other steps perform validation procedures, which verify data on the flash device without changing it. The *blank* programming step, for example, is used to ensure that the flash device is blank after erasure.

NOTE

Some manufacturers ship blank flash devices. However, every flash device should be verified as blank before programming. If the device is not erased completely, it cannot be correctly programmed.

For a complete list of flash tests and descriptions of the logical programming steps they perform, refer to Table I below:

Table I Flash Test Functions

Flash Programming Step	IPG Generated Test	Flash Library Model Function
id	*Check ID:* Filename: **"device number:id"** **(e.g. u1:id)**	This test uses embedded algorithms to verify that the manufacturing code on the flash device matches the expectation set up in the "board" file. Run this test on flash devices to ensure that the correct part is installed on the board. This test is typically used in connection testing in the digital subroutine.
erase	*Erase Device:* Filename: **"device number:erase"** **(e.g. u8:erase)**	This test uses embedded algorithms to erase the memory spaces available on the flash device. When erasure is completed, all memory cells on the device should read as FF. (On some early generation flash devices, erased memory cells on a blank device read as zeros.) If the part has not been erased, correct programming cannot occur.

Table I Flash Test Functions

Flash Programming Step	IPG Generated Test	Flash Library Model Function
blank	*Verify Erase:* Filename: **"device number:blank"** **(e.g. u11:blank)**	This test reads address locations on the flash device to verify that they have been erased correctly. Typically, this step is done before the *"program"* step to ensure the flash device is blank and ready for programming.
program	*Program Device:* Filename: **"device number:program"** **(e.g. u4:program)**	This test programs the flash device with hexadecimal data provided by Motorola® S-Records or Intel® Hexadecimal records. In addition, HP 3070 utilizes any intelligent device features such as embedded algorithms which verify successful programming routines.
verify	*Verify Program:* **"device number:verify"** **(e.g. u12:verify)**	This test verifies that programming has been accurately completed by comparing the data cells of the programmed device to the data source file used for programming. The verify routine is typically used only in the debugging process, and is seldom used in production testing.
crc	*Verify Data with CRC:* **"part number:crc"** **(e.g. u10:crc)**	The Cyclical Redundancy Check compares the data on the device with the data from a known good board. It performs the same function as the "blank" programming step, except it compresses the known good data into the "crc" object file. This test performs a CRC for each 2 channel part in the data field.

Figure 6-1 Digital Flash Test File Description

6.3 Locating Flash PDL and Test Directories

HP 3070 systems include standard flash PDL and VCL library models. You can find the standard flash library models in the following directories:

6.3.1 /hp3070/libraries/supplemental/flash

The PDLs that appear in this directory are named after the flash device part number with no extension. The device part number in the "board" file reflects these names.

Examples:

```
am29f040
am29sl800b
28F160
28F032
AT29c010
```

For flash programming, there are six library models referenced by the flash specific PDLs. The file names for digital library models begin with the part number of the flash device followed by an underscore and the name of the flash function that the test performs. For flash test descriptions, see Figure 6-1

Examples:

```
am29f040_id
am29f040_erase
am29f040_blank

28F160_id
28F160_erase
28F160_blank
```

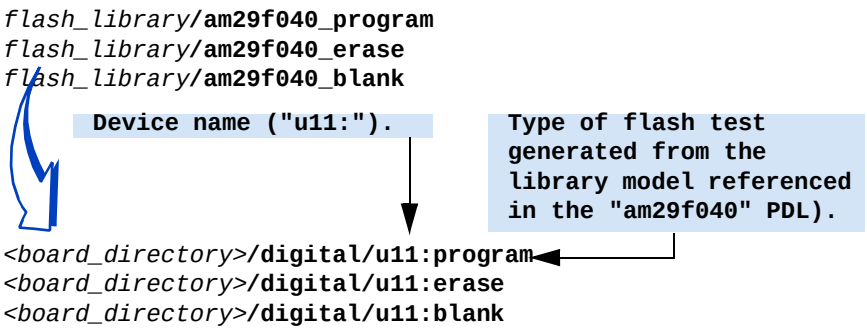
6.3.2 /hp3070/boards/*board_directory*/digital

This directory is empty until you have run IPG on your board (see Chapter 7, “Validating Tests for Production”). After running IPG, this directory becomes the target destination for digital flash tests identified in the "board" file. IPG generates flash tests by utilizing standard digital library models (e.g. "am29f040_id", "28F160_id", etc.). Unless instructed to generate tests selectively, IPG generates

six separate **VCL** tests for each flash device named in your "board" file.

The filenames for the flash library tests are created by combining the device name referenced in the "board" file and the flash function the test performs, as defined by the device name in the PDL.

Examples for the "am29f040" flash device might look similar to the following:



These tests will be modified to add disable statements, add hardware wait pins or add nodes to control other flash devices for multiple chip programming.

6.3.3 OBP Production Programming Task Flow

Flash70

Speed enhancements can be achieved by using the Flash70 solutions presented in **Chapter 8, "Series and Parallel Programming."**

The six library test models per flash device that IPG places in the "digital/" directory are used at some point during test development. The only required step for flash programming, however, is "*program*" (and "erase" if the part is not all ready blank). Often, unneeded library test models are not executed in the production test suite to save manufacturing time.

It is important to think about flash OBP programming tasks in two distinct phases: test development and production. The first phase, test development, involves designing effective tests for your manufacturing requirements; whereas the production phase is used to program flash parts and verify the data is correct during board test.

Once the test suite has been developed, production test strategies attempt to reduce the manufacturing time it takes to complete the programming during board test.This is usually accomplished by excluding tests like "verify" from the production test suite. For example, a **CRC** test could be substituted for "verify" in production. CRC tests are much faster and can effectively verify flash program accuracy, providing the test suite has been developed correctly. The

check ID test that appears in Figure 6-5, **HP Board Consultant Check List** is useful for **ICT** to ensure that the flash device installed is the one that appears in the "board" file.This test is used in the standard digital execution section to find manufacturing faults.

6.3.4 Flash Programming Test Flow

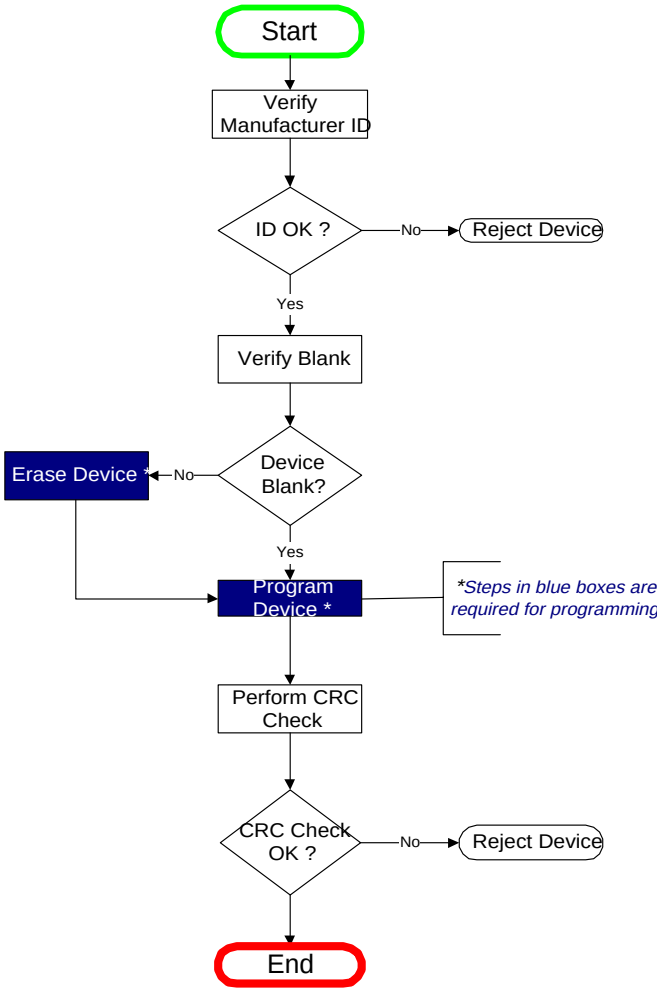


Figure 6-2 Flash Programming Flow Chart for Production

NOTE

To incorporate the CRC test as shown in Figure 6-2, **Flash Programming Flow Chart for Production**, a known-good board with programmed flash devices is needed. Ideally, the known-good board flash devices should be programmed by a mechanism other than the "program" test to ensure reliable CRC verification.

6.3.5 Using HP 3070 Libraries to Develop Flash Tests

To develop tests for flash devices, you can utilize existing HP 3070 libraries. Flash70 software provides Part Description Library (PDL's) models for programming the most common types of flash memory. A Part Description Library model consists of a suite of tests, each designed to perform a specific flash test function. You can use these library models with the corresponding flash device types. Or, you can easily adapt an existing library model to work with a device not contained in the library. You can access these libraries in the following directory:

/hp3070/library/supplemental/flash

6.3.6 Part Description Library Structure

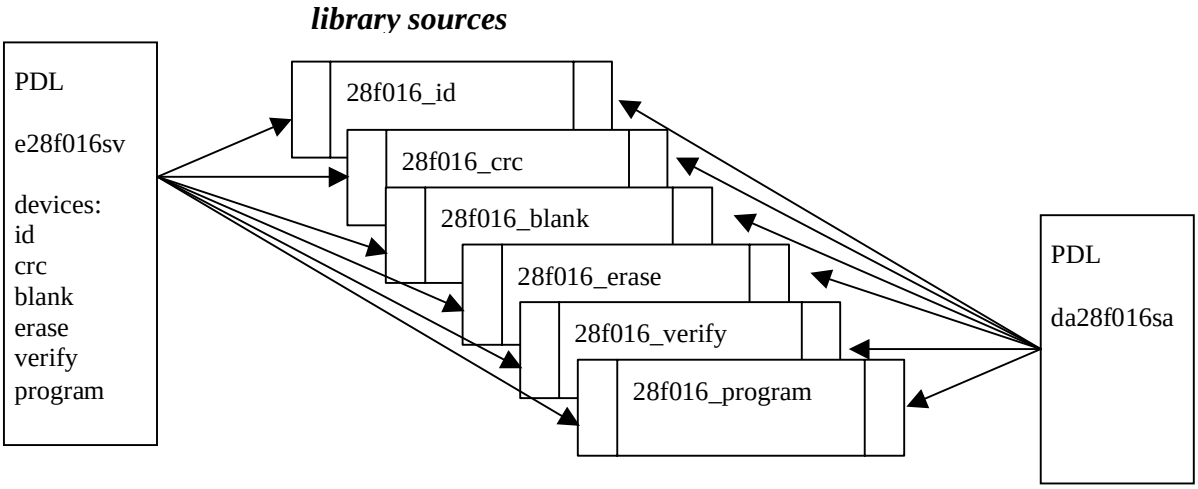


Figure 6-3 Part Description Library Structure

The Part Description Library (PDL) has the name of the individual device as it would appear in a bill of material. This name includes the full device designation, for instance **e28f016sv**. The PDL defines the individual tests to be used for this device. Since the tests stay the same for all versions of the same device type with only the pin out changing, several PDLs would point to the same tests. This way, different packages have different PDLs, but the same libraries. This vastly improves the efficiency of library storage and maintenance.

6.3.6.1 PDL's Features

- Header defining the device including:
  - Generic Name:device type used for libraries, i.e., 28f016 for all versions
  - Manufacturer: IC vendor
  - Packaging Type:tsop, ssop, etc.
  - Safeguard Type:HP 3070 safeguard options
  - Family Type:Board file threshold requirement
  - Description:Data to be added to master list (see master list)
  - Instructions:Information about utilization of flash
  - Part numbers identifying the libraries provided
  - Device names matching the test function, i.e. crc, id, etc.
  - "No safeguard" designation for all devices but one, to minimize warning messages
  - Pin assignments to all libraries for all pins including NC
  - "No Warranty" disclaimer

6.3.7 Library Structure

The libraries called by the part description library follow a standard format. This standardization simplfiies the process of modifying, implementing, and debugging library tests. Library tests are written to be utilized by the HP 3070's test generation software.

6.3.8 IPG, PDLs, and Flash Test Library Models

HP 3070 Series 3 uses the same PDL and test file implementation as earlier versions of HP 3070 software. However, PDL files are applied to digital flash devices in a different way. With the new flash library models, IPG Test Consultant now recognizes six digital "pin library" subdevices within one flash part. Since each subdevice is declared testable in the PDL file, flash device programming can be done in-circuit, in the same manner as other device tests.

For example, to program a flash device, you should identify, erase, and then program the appropriate hexadecimal data. The HP 3070

software generates a test for each pin library "subdevice" of the flash device as shown in the Part Description Editor below:

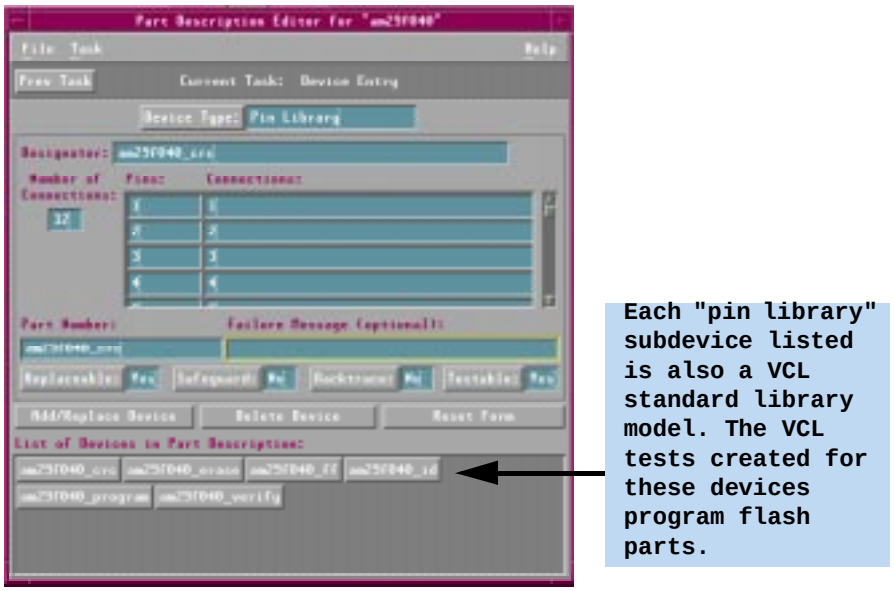


Figure 6-4 Library models as flash subdevices

The flash library models listed as subdevices in Figure 6-4, **Library models as flash subdevices** are used to create digital flash tests in the following manner:

1. IPG uses the PDL file that shares the same name as the flash devices listed in the "board" file as a model for the flash **OBP** digital tests, as shown in the example below:

   ```
   pin library "crc", ns, pn"am29f040_crc"
   pin library "erase", ns, pn"am29f040_erase"
   pin library "blank", ns, pn"am29f040_blank"
   pin library "id", ns, pn"am29f040_id"
   pin library "program",ns, pn"am29f040_program"
   pin library "verify",ns, pn"am29f040_verify"
   ```

   Since each flash library model referenced by the **PDL** is interpreted by **IPG** as a subdevice within the larger "am29f040" part, the flash functions (identification, erase, program, etc.) are executed when the test is run on the in-circuit flash device.

2. IPG builds digital test libraries for each flash device named in the "board" file.

For example, **`u11:erase`** is the digital test that erases the "U11" instance of an "am29f040" flash device on your board. In addition to

the erase test, IPG creates the following tests for the flash device named "u11": **`ID Check`**, **`Verify Erase`**, **`Program`**, **`Verify Program`**, and **`CRC Check`**.

6.4 Section Two: Steps to Developing Flash Digital Tests

The rest of this chapter outlines the steps you should take in using the HP 3070 software tools to generate the flash programming test suite. After completing test generation with IPG, you need to verify that the tests program the correct data onto flash devices. This procedure is described in **Chapter 7, "Validating Tests for Production."**

6.4.1 Step 1: Configuring the board 'config' file

Before using Flash70 features on a board, you must install the system codeword for Flash70 and enable Flash70 in the board "config" file. Enabling Flash70 activates HP 3070 software features that speed up the programming mechanisms used in the digital executable tests. Since Quick70 software and ControlXT cards are required for Flash70, these features must also be enabled in the board "config" file. To enable Flash70, the following syntax must appear in the board "config" file and one test library:

```
enable flash70.
```

NOTE

If you wish to utilize the automated Flash70 features in your test program, you must enable Flash70 in the board "config" file. If you "enable flash70" in the test suite and not in the board "config" file, IPG generates warnings and places them in the OBP tests.

6.4.2 Step 2: Verifying IPG Test Generation

For flash programming, HP Board Consultant is used primarily as a verification tool. Key HP Board Consultant tasks include verifying library paths, disabling methods, and that safeguards have been turned off so flash devices can be programmed quickly and efficiently. The primary tasks are shown in Figure 6-5:

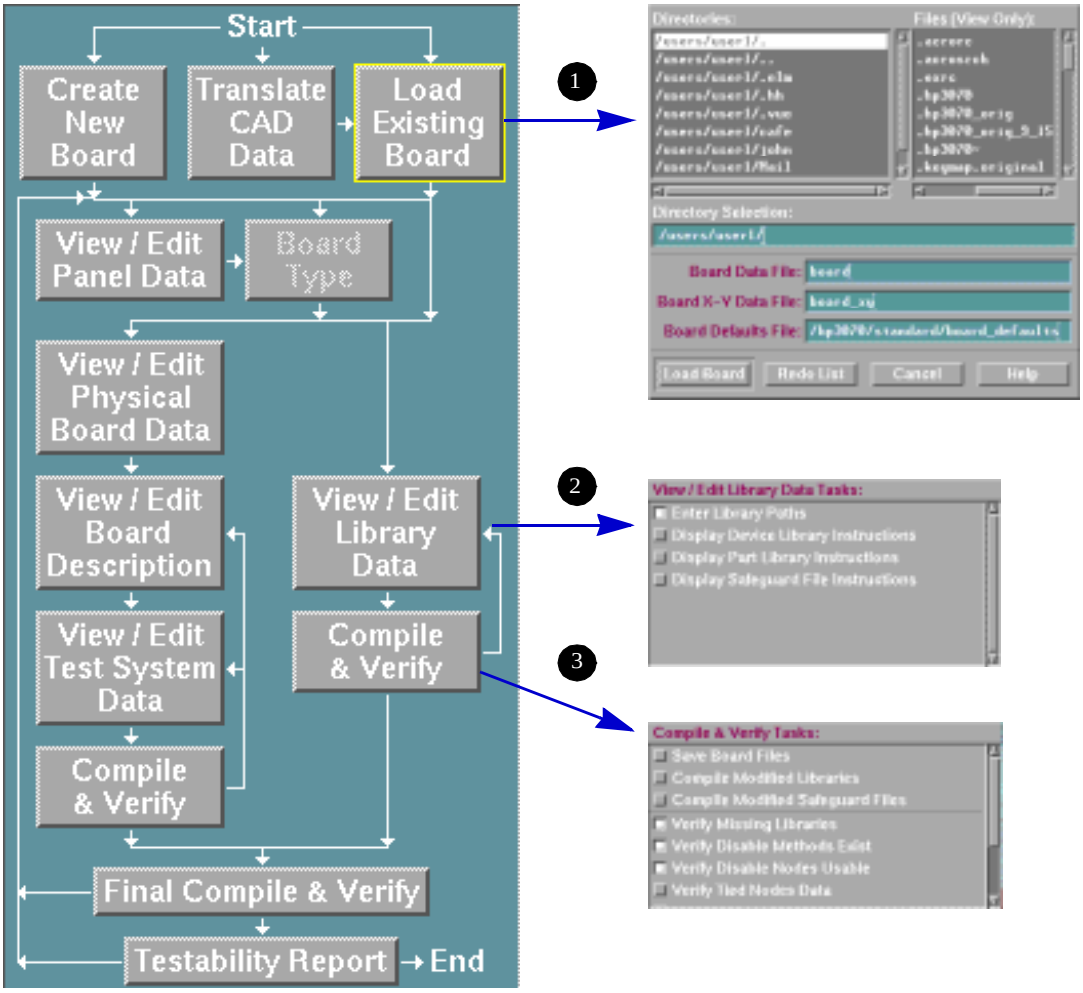


Figure 6-5 HP Board Consultant Check List

To set up your board for flash OBP in HP Board Consultant, complete these steps:

1. Load the board containing the flash devices into HP Board Consultant.

Load a board and run HP Board Consultant by dragging the folder containing the board files onto the "HP Board Consultant" icon:

Drag and Drop Shortcut:

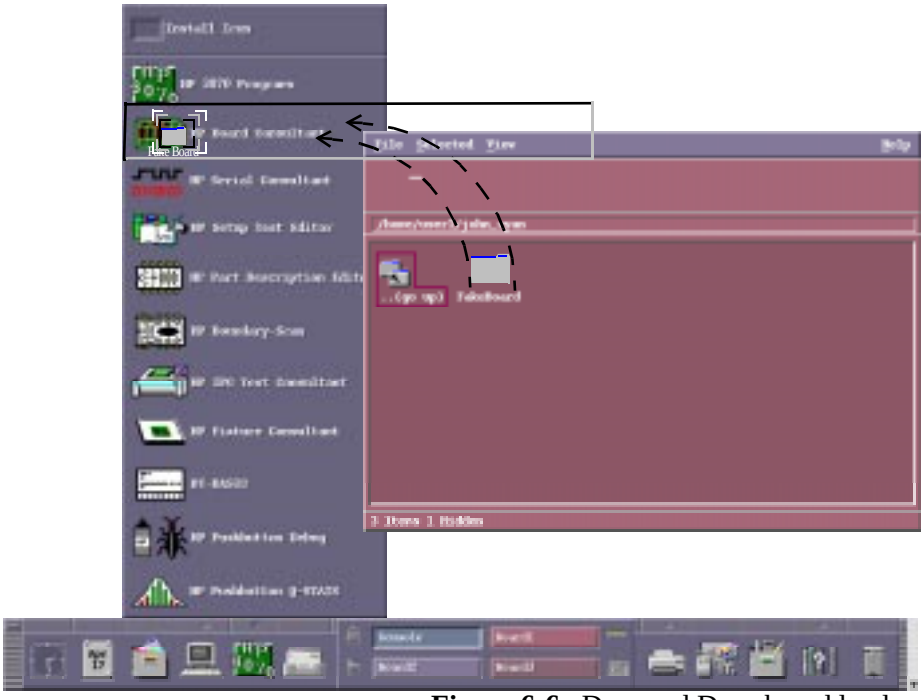


Figure 6-6 Drag and Drop board load

2. If you have created digital flash library models for devices not represented in HP 3070 , enter the directories that contain your custom flash device library files into the Library Paths formLibrary Paths Form.

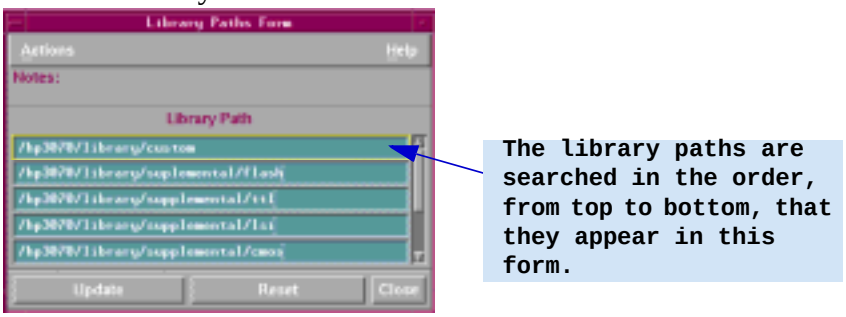


Figure 6-7 Library Paths Test Form

Since the board compiler searches library directories in the same order that they are listed in the Library Paths form, the directories containing custom PDLs should be entered into the list before the paths that contain the standard HP 3070 flash

device libraries. Set the flash device number to the appropriate PDL name. Verify that library test options have been set correctly for all the flash devices on your board.

Open the Device Entry forms for your flash part subdevices.

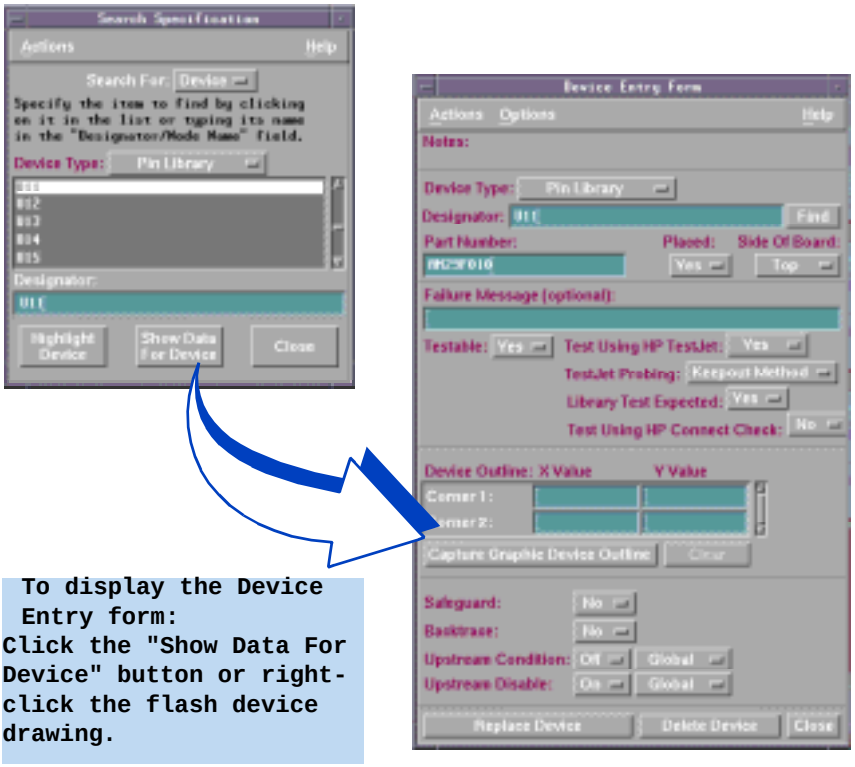


Figure 6-8 Device Entry Form for a flash part

3. From the Device Entry form, do the following:

- Verify that the library test points to a valid PDL name in /hp3070/library/supplemental/flash or your custom library directory (see Figure 6-8, **Device Entry Form for a flash part**).

- Confirm that the "Testability:" and "Library Test Expected:" fields are set to "yes" for each device ..

IMPORTANT

If the board under test was not intended for on-board programming, consult with the test design team to verify that tests eliminate damage potential from excessive backdriving.

***For information on design considerations for on-board programming, see* Chapter 2, "Design For On-board Programming."**

4. After all the appropriate libraries and device options have been set and verified, compile the tests.

Figure 6-9 Final Compile of board files

NOTE

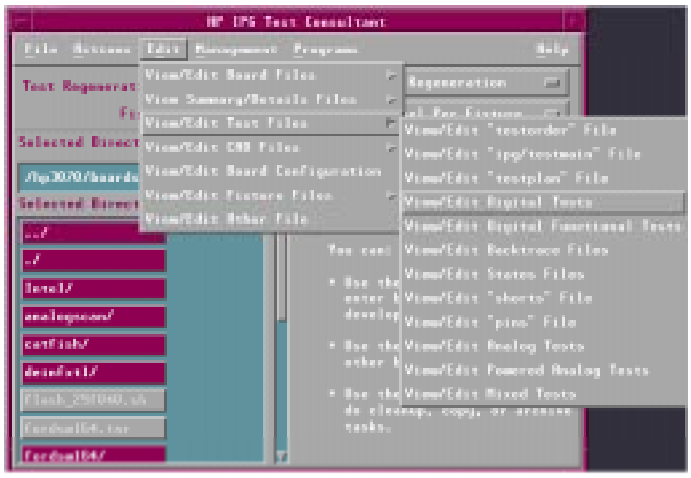
For detailed information on creating board files and part description libraries, see **Test & Fixture Development**

6.4.3 Step 3: Running HP Test Consultant

IPG generates six VCL tests that correspond to the subdevices of the flash parts installed on your board. After the tests have been generated, you need to open at least one of the flash VCL test files in "digital/" to ensure that all requirements for OBP are included. To verify that the newly created flash test suite is in working order, open the "program" tests for each type of flash device installed.

Verification steps follow:

1. From the HP Test Consultant, select **Edit** *"View/Edit Test Files" "View/Edit Digital Tests"* from the HP pulldown menus as shown below:



This selection launches BTBasic and switches to the "digital/" subdirectory of your board directory.

2. Load the test files for each flash device into the BTBasic window as you normally would (e.g. enter "get u8:program" on the BTBasic command line).

3. Look for unexpected disabling warnings.

   Six disabling warnings for every flash test installed on your board will be listed in the tests.

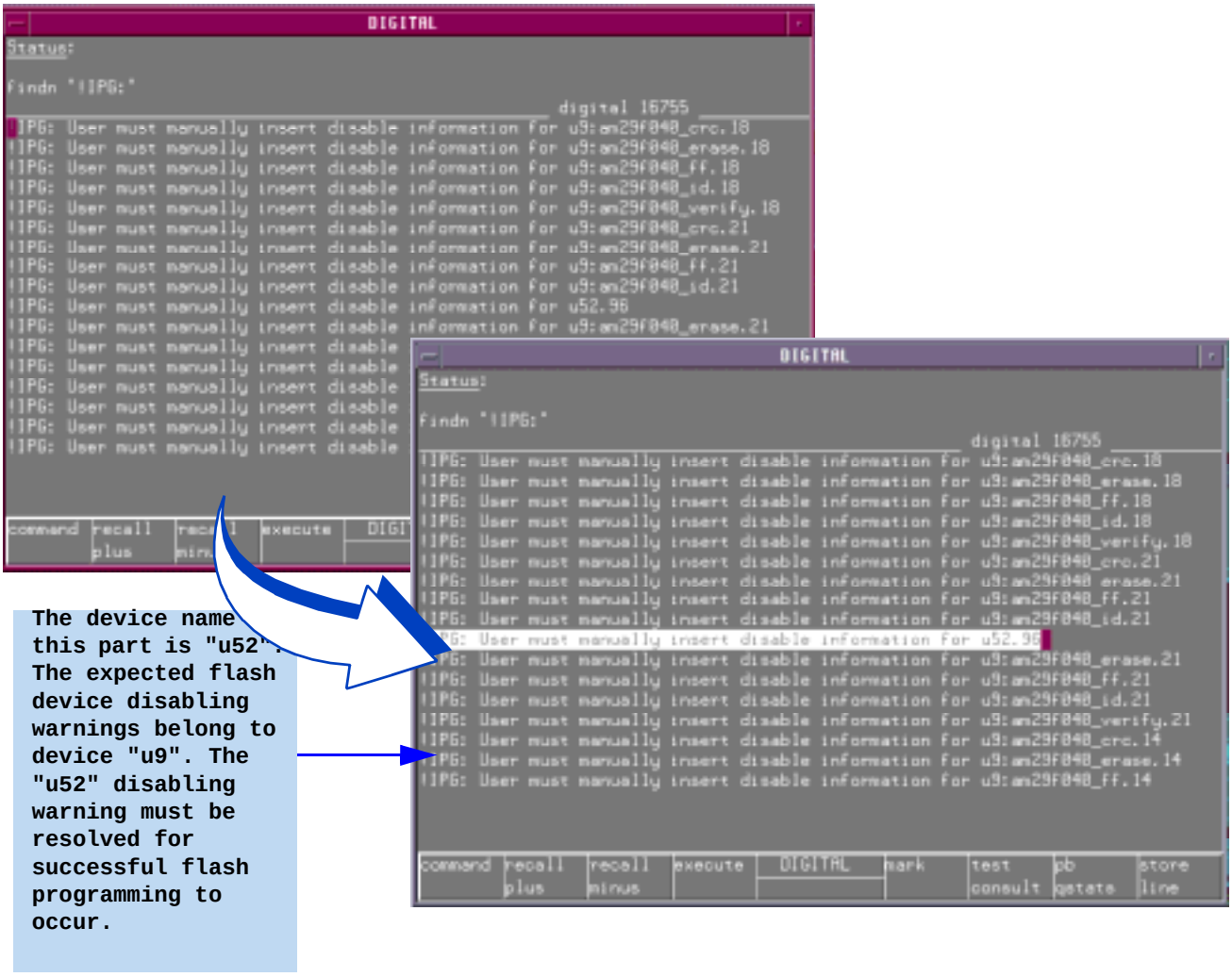
   Remember, the standard flash libraries are not what IPG would normally expect. They are only logical flash programming steps not "pin library" subdevices. Since these devices share the same pin assignments, IPG reports that they must be disabled for testing. In fact, they do not need to be disabled, and any disabling warnings about the flash devices that share same device name as your installed part can be ignored.

   ♦ How to search for important disabling warnings.

   You can find disabling problems by searching for character strings that begin with "!IPG:" and reading through the device names shown. Once the warnings are found, any device name that does not belong to the flash device under test is suspect.

   Since the IPG warnings are written to every test file for the flash device, only one test file must be searched in this manner. However, any disabling which the test developer determines to be required must be added to each test in the suite.

The process is shown in the BTBasic window below:



NOTE

Highlighing is added here for demonstration purposes, so manual searches of the disabling warnings are necessary.

Figure 6-10 Disabling warnings in BTBasic

4. Some devices can use hardware waits to speed up programming. There will be commented instructions in the library in these cases, uncomment the wait assign statements. If you have purchased combinational testing (i.e. enable combo appears in your "config" file) or the Flash70 package, then comment out the homingloop waits in your VCL "program" and "verify" tests and uncomment the wait assign statements.

The assignment section should have the following syntax:

```
!ASSIGNMENT SECTION
wait line STS
wait terminated when STS is "1"

homing loop 60000 times        ! allow cell to program
 execute Device_Ready exit if pass  ! Program Completed
end homingloop

! execute three_state wait
```

Modify by commenting out the homing loop:

```
!ASSIGNMENT SECTION
 wait line STS
 wait terminated when STS is "1"



! homing loop 60000 times       ! allow cell to program
!  execute Device_Ready exit if pass  ! Program Completed
! end homingloop

 execute three_state_wait
```

NOTE

You can improve flash programming speed by approximately 15 percent by making the above changes.

6.4.4 Power Voltage Considerations

Often flash devices use special power levels for programming, usually 12V is applied to V_{PP}. The V_{PP} pin on the device should be supplied with a separate power supply. In the board file, this power supply should be set to the standard V_{CC} level of the board. The user will modify the testplan power-up subroutine for flash programming. The power up sequence should first bring the device to standard V_{CC} levels for V_{PP}. After this, the V_{PP} level may be set. Some devices have been known to have programming failures if the power supply is taken directly to the V_{PP} 12V level.

Programming voltage levels need to match the final operating voltage of the board. Automatic algorithms make this necessary since they validate the programmed voltage against the threshold dictated by V_{CC}. If the device is programmed at a lower voltage level than it will operate, the "1" and "0" states may be lower than required in operation. To avoid this problem, use the voltage regulator on the board to control the power supplied to the device during

programming. If the V_{CC} levels are supplied directly, through an edge connector node, the testplan should set V_{CC} to a level at or above the level supplied to the board in operation.

If your board file compiles properly with part description libraries assigned for the flash devices, HP Test Consultant can be executed. HP Test Consultant generates the flash programs automatically. The correct tests will be placed throughout the board directory in the appropriate locations. If you have modified the digital tests, rerun HP Test Consultant so new wires will be added to the fixture "wirelist".

CautionAlways backup your tests before making any modifications.

6.4.5 Step 4: Modifying the ’testplan’

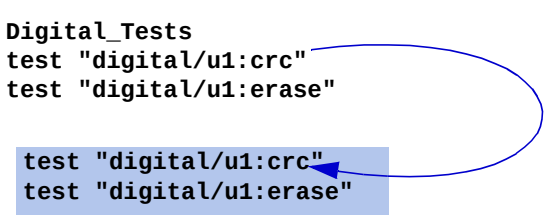
The testplan generator (TPG), generates the "testplan" file. The testplan is comprised of a testmain that controls the testing of the board, and subroutines that contain the statements to execute the device tests:

Although the "testplan" contains the flash subroutines necessary to run the proper sequence of test steps, you must modify the "testplan" slightly for flash programming. The instructions are described within the BT-Basic "testmain" or "testplan above the sub Program_Flash. Since instructions for this vary from part to part, you must read the instructions provided in the "testplan" for more information on this process.

6.4.5.1 Modifying the "testplan" for Panel or Throughput Multiplier Topologies

1. Copy the test statements for your flash devices from the "Digital_Tests" subroutine to the "Program_Flash" subroutine. Find the sample test in the program flash subroutine and replace it with the line from the IPG generated "testplan".

   The syntax under the "Digital_Tests" subroutine will look similar to the following example:



2. For flash devices needing a 12V power supply for programming, add these two lines to the "Setup_Power_Supplies" section in the "testplan":

```
setup_power_supplies
   if flash_program$ = 1
   then wait 2us !*! cps  ! raise voltage to 12 from initial
setting after wait
else
```

NOTE

There is a wait included to change the V_{PP} level from 5V to 12V as recommended by some manufacturers.

To modify the "testplan" for standard board topologies:

1. Enter the parameters for the device names by modifying the "Device_New$" lines in the "testplan".

   Since instructions for this vary from part to part, you must read the instructions provided in the "testplan" for more information on this process.

2. If multiple devices are programmed, simply add a new Device_New$ and call statement to the testplan.

3. For flash devices needing a 12V power supply for programming, the power supply will be raised to 12v after all power supplies are first set to normal operating voltages and the device is ready for programming.

```
setup_power_supplies
   if flash_program$ = 1
   then wait 2us !*! cps  ! raise voltage to 12 from initial
setting after wait
else
```

NOTE

There is a wait included to change the V_{PP} level from 5V to 12V as recommended by some manufacturers.

4. In some programming cases, the power supplies need to be changed from operating voltage to programming voltage.

   After programming, however, power supplies will be reset to operating voltage.

6.4.5.2 Other "testplan" considerations

Some of the following board considerations must also be addressed in the "testplan" and "testmain" file:

- The need to cycle power between digital test and flash to put volatile **FPGA** back to three-state mode
- Power supply setup should have an additional copy that increases the V_{PP} power supply
- Non-blank status results in erase of all parts on same write out execution (rework boards reprogram) unless the user modifies this operation.
- Make sure cluster is in board at onset if parallel programming is an option

6.5 Section Three: Flash Tests and Existing Fixtures

Developing flash programming tests as shown in **Section One: Flash OBP Programming Steps, on page 6-5** utilizes the standard flash features of B.03.00 software. If you have existing digital files that you want to use for flash programming tests, you can add the new flash features by modifying the "board", "testorder," and "wirelist" files. The modifications involved are minor and will not require a new testplan.

NOTE

Always backup your files before making modifications.

To achieve optimal programming speeds, you must set up your tests with dynamic pin assignments or use Flash70.

The digital test update procedure follows:

1. Back up your existing "board" directory.

2. Open HP Board Consultant. (e.g. Drag and drop the board directory onto the HP Board Consultant icon (see Figure 6-6, **Drag and Drop board load**)).

3. Add the path "/hp3070/supplemental/flash" to the Library Path Options section in the "board" file.

   The Library Options section of the "board" file should look similar to the following:

```
LIBRARY OPTIONS
"/var/hp3070/library/supplemental/flash"
"/var/hp3070/library/ttl"
"/var/hp3070/library/lsi"
"/var/hp3070/library/cmos"
.
.
.
```

Copy this line from the "board_defaults" file into your local "board" file.

- To change the library paths in the "board" file, use HP Board Consultant to select the "View / Edit Library Data" flow chart icon, and then click "Enter Library Paths"):

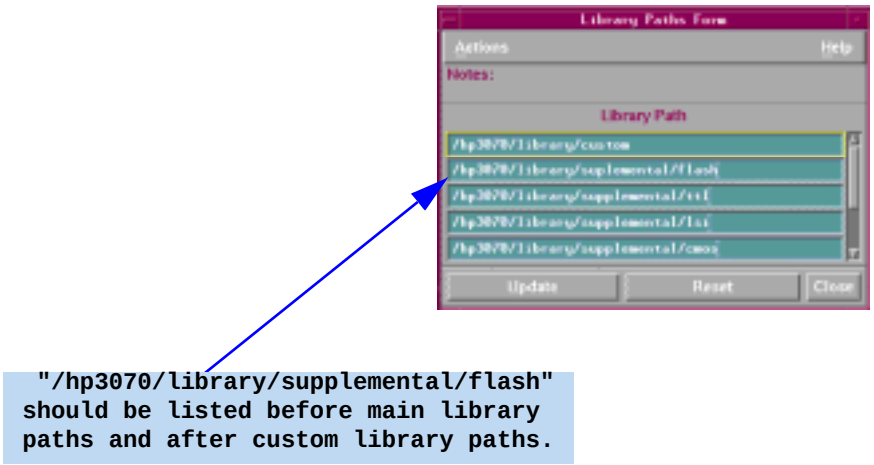


Figure 6-11 New board_defaults Library Path form

4. Enter the appropriate flash Family Options in "board" file.

   The easiest way to accomplish this is to open the "/HP3070/library/standard/board_defaults" file and copy the Family Options flash family descriptions into your local "board" file.

   The available flash Family Options are displayed below:

```
FAMILY OPTIONS
.
.
.
FLASH_5V
      Drive High     4;
      Drive Low     0.2;
      Receive High   2.4;
      Receive Low    .5;
      Edge Speed     50;
      Open Input Default X;
      Load    UP;

FLASH_3V3
      Drive High    2.5;
      Drive Low     0.2;
      Receive High   2.0;
      Receive Low    .3;
      Edge Speed     50;
      Open Input Default X;
      Load    UP;
```

```
FLASH_2V2
      Drive High    2.0;
      Drive Low     0.2;
      Receive High   1.85;
      Receive Low    .4;
      Edge Speed     50;
      Open Input Default X;
      Load    UP;

FLASH_3V3
      Drive High    1.7;
      Drive Low     0.2;
      Receive High   1.25;
      Receive Low    .25;
      Edge Speed     50;
      Open Input Default X;
      Load    UP;
```

- After adding flash families to the "board" file in HP Board Consultant, you can view the results by selecting the "View / Edit Test System" task flow icon, and then clicking "Enter Family Options.":

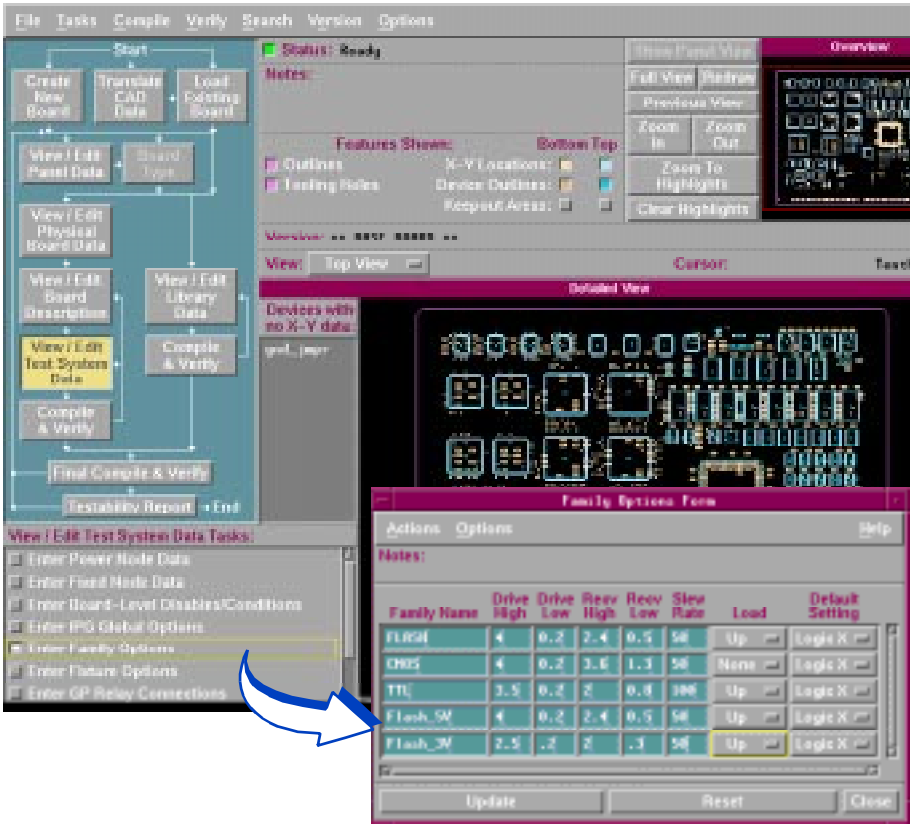


Figure 6-12 ECO flash family addition

5. To retain the customization you've entered into your "testplan" file, make sure that the "Testplan generation off" statement is not commented out in the "testorder" file:

6. Make sure the flash tests listed in the "testorder" file are not marked "permanent", so that IPG will generate new digital tests for your flash devices,

7. Change the part numbers in the "board" file to match the new PDL libraries.

NOTE

If you do not want to change the name of the PDL to the device name seen in the board file, you can add the new test suite to the boards custom library directory.

8. Run a comprehensive regeneration on the test suite in **HP IPG Test Consultant**.

   This procedure is best done in incremental mode, so you can deal with any unexpected problems as they arise.

9. Follow the instructions found with the sub program_flash in the appropriate testmain found under "/hp3070/standard

6.6 Set Up A Flash Test Suite to Validate Your Test

To set up your newly created flash test suite, see **Chapter 7, "Validating Tests for Production."**

Chapter 7
Validating Tests for Production

Rev. A

IMPORTANT

BEFORE YOU BEGIN

We recommend that you setup flash OBP test suites only after all other board tests have been written and debugged. If problems occur in the other board tests, flash tests can be extremely difficult to perform .

For best results, follow the procedures in this chapter. This will save time in preparing the test suite for the production board test environment.

Test developers can use the information in this chapter to implement the setup process and troubleshoot failing tests.

This chapter requires an understanding of the following:

- Flash programming fundamentals. See **Chapter 1, "Introduction to Flash Programming."**
- Design considerations for on-board programming. See **Chapter 2, "Design For On-board Programming."**
- How to generate flash digital libraries for the board under test. See **Chapter 6, "Generating Flash Digital Tests."**

This chapter describes:

- The OBP flash device test suite task flow. See **Task Flow for Testing OBP Libraries, on page 7-2**.
- How to set up the OBP flash device test suite. See **Using IPG Generated Flash Tests to Setup OBP, on page 7-4**.
- Comparing the HP3070 B.3.00 programmed data with the data on a known-good device. See **Verifying that Programmed Data is Correct, on page 7-21**.

7.1 Overview

The setup process for flash on-board programming is used to validate tests before they go to production. Setup involves executing a series of tests to validate that programmed data matches the data results expected from the user defined data source. These tests help to determine if the HP 3070 programming mechanism works with the data that board designers have provided to tests developers.

Setting up flash programming tests is easy because the HP 3070 provides standard libraries for the most common flash devices. These libraries require minimal debugging. However, it is often necessary to modify existing test files to accommodate variances in board design. This chapter discusses the flash programming test flow used to validate tests for production. It also addresses potential problems you might experience and some common causes for OBP test failure.

7.1 Task Flow for Testing OBP Libraries

Board designers should provide test developers with a known good board that contains pre-programmed flash devices. HP 3070 flash tests enable test developers to view the actual data of a known-good flash device before starting on-board programming. If the flash tests need to be modified, knowledge of what the data *should* look like is invaluable. If the board designers do not provide pre-loaded flash devices, OBP test setup can still be accomplished, but the process is more difficult to implement.

As described in **"Locating Flash PDL and Test Directories,"** the HP 3070 flash software provides PDLs and library models that enable flash on-board programming (see **Table IFlash Test Functions** for library descriptions.) There are six library models for each flash device installed on your board. IPG uses these library models to generate a test suite that consists of six files. These tests are used to verify that the flash devices installed on your board can be correctly programmed with the data provided. If there are flash devices that do not have standard libraries, the OBP setup procedure can also be used to debug custom libraries.

IMPORTANT

Setting up flash test libraries for in-circuit OBP is dependent on upstream board devices functioning properly. Therefore, it is important to debug all other board tests before setting up flash OBP test suites.

7.1.1 Setup Process Task Flow

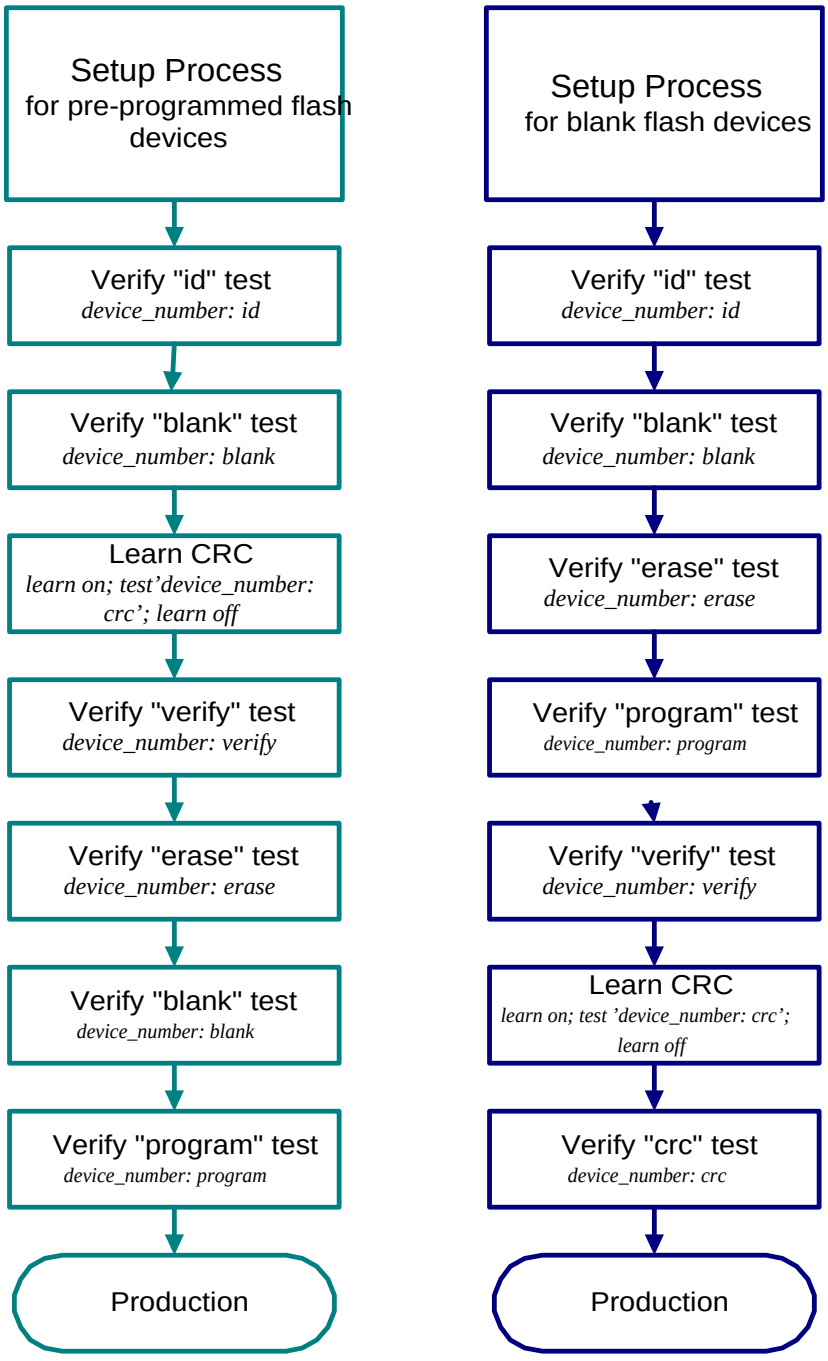


Figure 7-1 Flash OBP Setup Process Using a Known Good Board

7.2 Using IPG Generated Flash Tests to Setup OBP

IMPORTANT

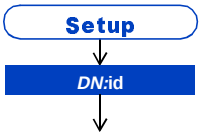
When boards are delivered with pre-programmed flash devices, a programming mechanism other than the "program" test should be used on the known good board flash devices. This practice provides the means to compare your programming results with known-good data. This is the best way to ensure reliable OBP test suites.

When boards are delivered with blank flash devices, its essential that the HP 3070 programming process can be verified by a method other than the flash test libraries. Using an alternative verification mechanism is the best way to ensure that the "program" test has programmed expected data. A functional test, for example, can be run to verify that the board works with the newly programmed parts.

Since flash OBP uses digital test libraries to program devices, you use the HP 3070 "debug" program to set up the flash test suite.Whether you use HP Pushbutton Debug or BT-Basic commands, the implementation process is the same. Flash tests that pass in debug, however, do not necessarily mean that the device is being programmed with correct data. It is important to keep in mind that flash OBP uses digital library "tests" to actually program the device.

The 'id' test, on page 7-5 and **The 'program' test, on page 7-15** sections of this chapter explain how to use the six flash tests to set-up and, if necessary, debug a problematic test suite. The following sections are presented in the order that test developers would normally follow for setting up pre-programmed on-board flash devices.

7.2.1 The ’id’ test



The identification test establishes that the correct device is installed and that the pin inter-connects on the board are working. The test reads the manufacturer code and device identification number from the memory chip. If the device utilizes CFI standards, the test might also verify additional register contacts that more fully test pin states for interconnect verification. Run the "id" test first in OBP test suites. If the test fails, it is likely that the part installed is not the one the test expected to read.

Possible causes for "id" test failure follow:

- The installed flash device is different than the device for which the flash test suite was developed.
- Pin assignments do not match datasheet specifications for the device.

  If this occurs, verify that the pin name matches the pin number shown in the data sheet.
- Upstream devices have not been disabled.

  Important IPG disable warnings can be buried inside the test file because flash device tests can’t disable themselves. For more information, see **Step 3: Running HP Test Consultant, on page 6-17**.
- Vpp and Vcc voltage levels have been incorrectly set for flash devices in the testplan.
- Thresholds are not correct for the device signal lines.

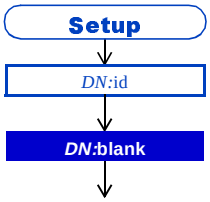
  Voltage levels for flash devices must be defined in the board defaults file. An example of voltage specifications for a flash family follows:

```
family FLASH
   Drive High 4;
   Drive Low 0.2;
   Receive High 2.4;
   Receive Low .5
   Edge Speed 50;
   Open Input Default X;
   LoadUP
```

- Timing variations in vector cycles, bus cycles, and offsets, making it impossible for the "id" test to read the device.

  Debug the ID test by comparing the graphic representation of control lines to the data sheet for the part.

7.2.2 The ’blank’ test



The blank test can be used to verify that a device contains pre-programmed data. The "blank" test verifies that every byte in the device is set to FF hexadecimal. Flash devices can only be programmed by changing bits from "1" to "0". After you have verified that the flash devices you want to program are installed correctly and the timing variations are set correctly, verify the device data by performing a "blank" test on the device. Use the flash library "blank" test.

The "blank" test is the simplest test in the flash OPB test suite, because it has no command structure. The test simply reads data from the device. The usual purpose of the "blank" test is to verify that a flash device has been erased properly by the "erase" test. On pre-programmed parts, however, the "blank" test can reveal potential problems with bus cycles. Understanding these problems is necessary during program development.

- For pre-programmed flash devices on a known good board, the "blank" test should show fail. If the test passes, then the board designers have provided a known good board with blank flash devices.

  After verifiying that the device is blank, proceed with the next step in the setup process. See **Task Flow for Testing OBP Libraries, on page 7-2**.

- When used on pre-programmed flash devices, the "blank" test should demonstrate the flash device *can* be read, and *appears* to be programmed.

- The blank test should show the expected data contents of a pre-programmed device when used in Pushbutton Debug. If the data matches the expected source data, the digital "program" test is working properly. If the data doesn’t match, some debugging may be required to correct control line problems.

  See **The ’id’ test, on page 7-5**.

7.2.2.1 Evaluating Device Data With Pushbutton Debug

The "blank" test is useful for many purposes when verifying operation of the "program" and "erase" tests. After verifying that the device is readable, you can evaluate data by utilizing "blank" test for pre-programmed devices. This can be useful when other tests in the suite fail.

To evaluate the programmed data with the "blank" test, you can use the display feature of "Pushbutton Debug." To display accurate addresses for the data, the pins displayed must be divisible by 4. This procedure enables you to read the data that is programmed on the memory device and then compare it to the data sheet.

Drag and Drop Shortcut:

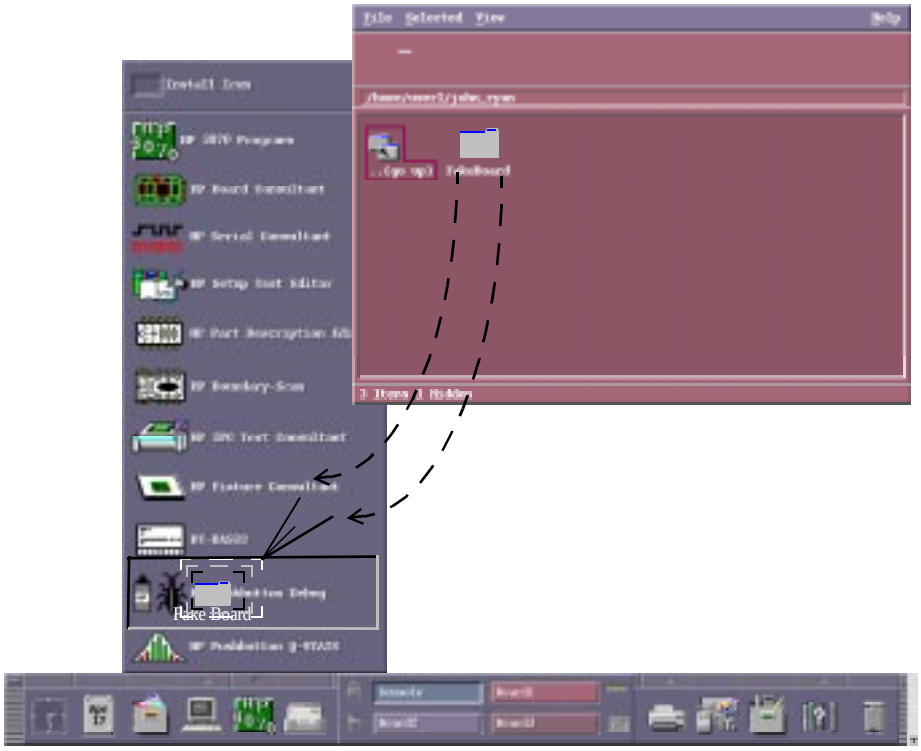


Figure 7-2 Drag and drop test initialization

The steps to display correct data addresses are described below:

1. Load the known good board.

   To load a board, start BT-Basic, and then run Pushbutton Debug. Drag the folder containing board files onto the Pushbutton Debug icon.

2. After loading the board, do the following:

   - Set the HP 3070 up for powered tests. Enter "powered" in the BT-Basic window.

   - Since Flash tests are time intensive if the safeguard feature is used, you must inhibit safeguards. To inhibit safeguards,. enter "safeguard none" in the BTBasic window.

   - If you use oscillator disables such as gprelays, run the portion of the "testplan" file that includes the power supplies, and any oscillator controls. An example follows:

```
unpowered
gpconnect 20654 to 20665
gpconnect 21856 to 21869
wait 20
powered
```


3. After the board is loaded into Debug and compiled, run the "blank" test. Enter "execute to fail" in the BT-Basic window.

4. Open the "Digital Debug Graphical Waveform" window and set the display to hexadecimal. Enter "display hex" in the BT-Basic window.

5. Display the data bus and address bus in hexadecimal format in one of the following ways:

   ♦ If the quantity of pins on your memory device address is divisible by 4, open the display groups. Enter "display groups *DATABUS, ADDRESS_BUS*" in the BTBasic window.

   ♦ If the quantity of address pins on your memory device is not divisible by 4, display data first. Then add or remove additional pins until the total number of pins showing are divisible by 4.

   Use Pushbutton Debug to pad the hexadecimal address display with extra control lines, or to create a display group in the test file that eliminates the high-order address pins. A description of these procedures follows.

7.2.2.2 Displaying Accurate Addresses by Adding Extra Control Lines

- Add as many control lines as necessary to display the data bus in multiples of four. For example, if you have an 18 address pin device, 2 extra control lines are needed to display the addresses correctly.

  In the diagram below, one extra control line is needed:

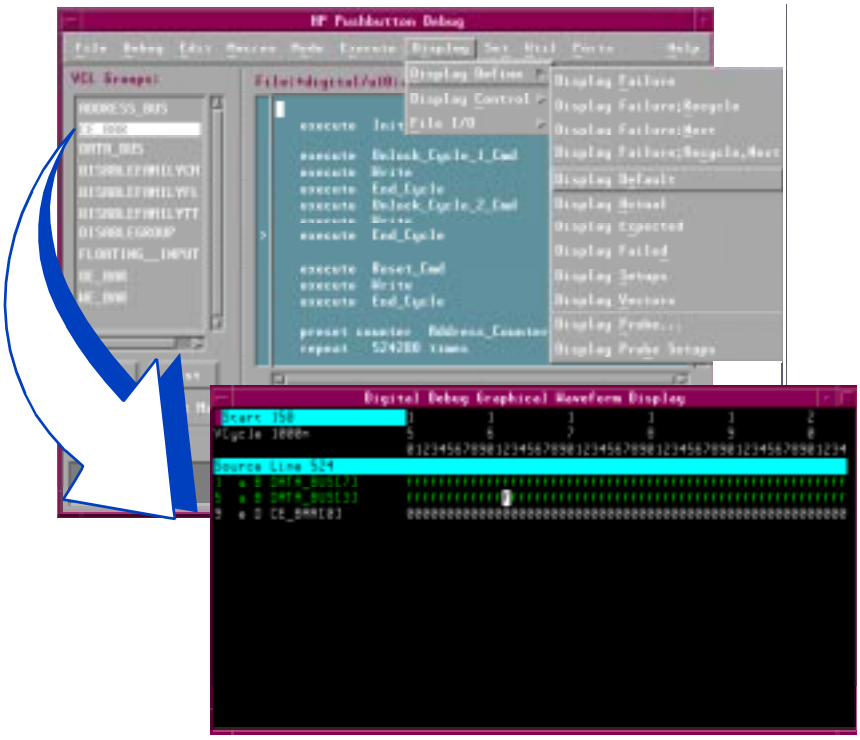
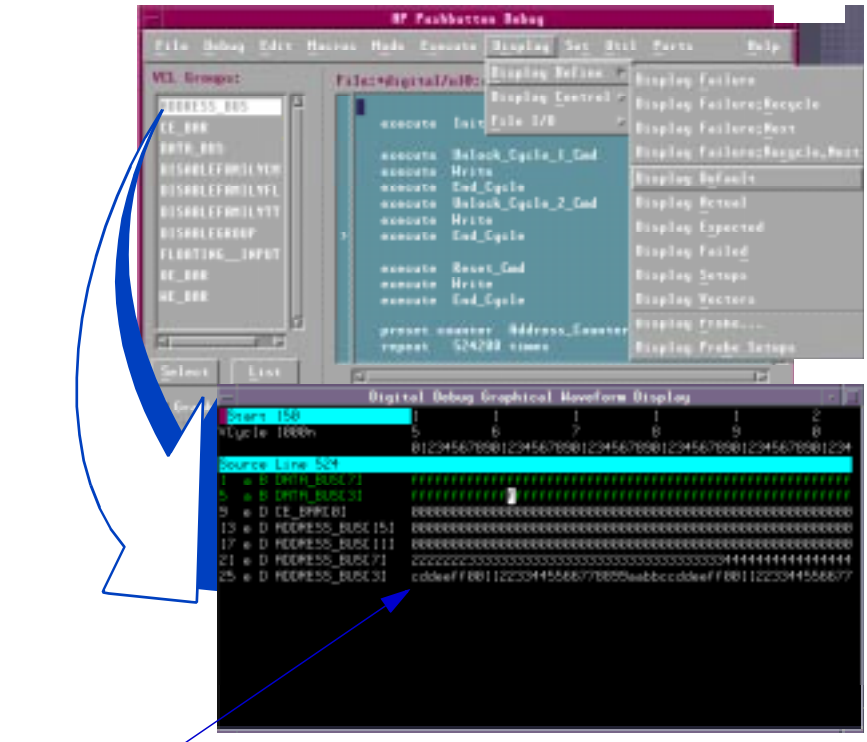


Figure 7-3 Padding the Hexadecimal Display with a Single Pin Control Line

The CE_BAR display group above has been assigned to a chip enable pin. Since the flash device's chip enable has a constant value of 0, it is an especially good control line to add to the data bus display as you will see in **Using the Hexadecimal Display to View Address Bus and Control Lines, on page 7-10**.

- After the appropriate number of control lines have been added to the waveform graphic display, renumber the display,

and then view it again in hexadecimal format. The result should look similar to the following:



The constant CE_BAR value of 0 makes the address easy to read.

Figure 7-4 Using the Hexadecimal Display to View Address Bus and Control Lines

The numbering of the address pins (0-3, 4-7, 8-11, 12-15) indicate the address bus is divided properly.

7.2.2.3 Displaying Accurate Addresses With the New Display Group

A better way to display addresses in 4 bit increments for data comparison is to change the "blank" test. This is easily done by copying the pin assignments of the address bus and leaving off the high-order pins. For example, if you have 19 pin device, you can create a new address group that only displays the first 16 pins as shown below:

```
assign  Address_bus to pins 1, 30, 2
assign  Address_bus to pins 3, 29, 28, 4, 25, 23, 26, 27
assign  Address_bus to pins 5, 6, 7, 8, 9, 10, 11, 12

assign  Address_john to pins 3, 29, 28, 4, 25, 23, 26, 27
assign  Address_john to pins 5, 6, 7, 8, 9, 10, 11, 12
```

Copy the pins from the original group to the new display group.

And then add the new display group to the display section of the test as follows:

```
inputs Address_bus, CE_bar, OE_bar, WE_bar, Address_john
```

After you have added the appropriate lines to the "blank" test, display the new control group after the data bus.

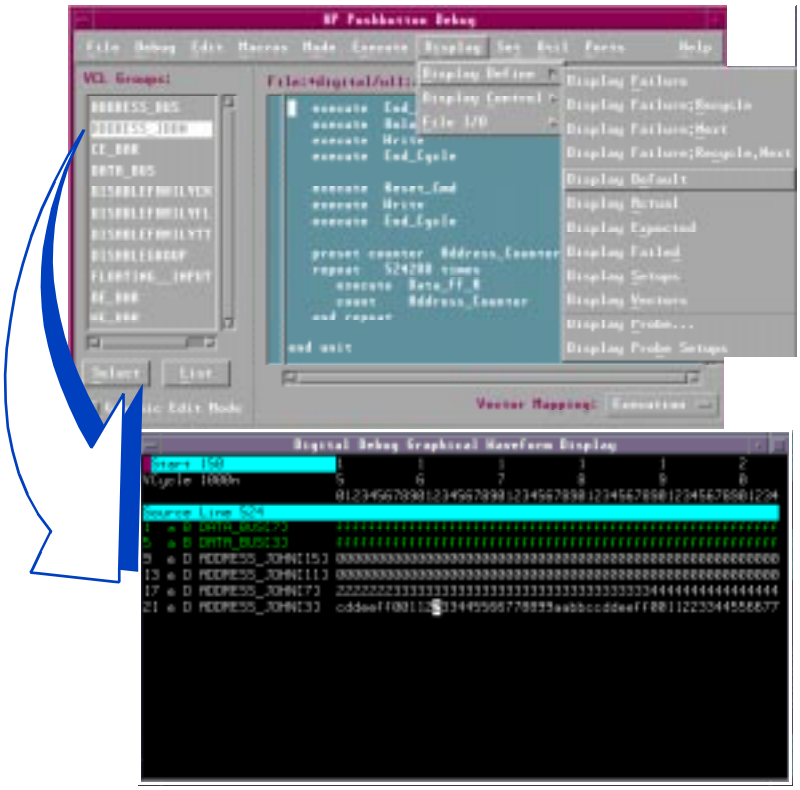
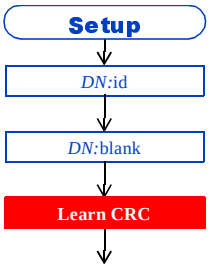


Figure 7-5 The Hexadecimal Display for a Truncated Address Bus

6. When the address bus display matches the data bus, compare the data at key addresses with the data sheet.

For example, to verify that the data is correct on the AMD part "AM29SL800B," display vector 4 to verify that it reads "555AA" for Word programming or "AAAAA" for Byte programming. If the data matches the data sheet command definitions, it is likely that the device is working properly and the other tests in the OBP suite should work as expected.

7.2.3 The 'crc' test



The "crc" test behaves like the "blank" test with a compression statement. It reads the data on the flash device in order to generate an algorithmic number based on the data residing on the chip. It then compares the result to the known-good cyclical redundancy check test. You can use the "crc" test in several ways. One of which follows:

If you are following the pre-programmed setup described in **Using the Hexadecimal Display to View Address Bus and Control Lines, on page 7-10**, the next step is to learn the **CRC** on a known good board and then store the CRC object file in a safe place. This file will be used to verify the programming step.

After running **"The 'id' test"** and **"The 'blank' test"** on a pre-programmed flash device, you have verified that the flash device installed on your known good board is the correct part and that the programmed data can be read and seems accurate based on the data sheet. Once these assumptions have been tested, learn the CRC.

To learn the CRC, enter **"learn on; test 'u2:crc'; learn off"** in the BT-Basic window. Then verify that the newly learned CRC works again on the known good board.

The "crc" check should pass. If it fails:

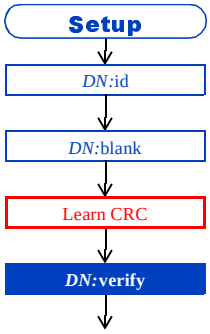
- Check for timing differences in receive delays between the "blank" test and the "crc" test.
- Check for offset differences between the "blank" test and the "crc" test.

IMPORTANT

It is extremely important to obtain a known good board programmed by a method other than the HP 3070 ATE. If the board has been programmed and debugged successfully by some other means, then the "crc" test should pass. If it fails, the problem is likely to be with the ATE connections, voltage levels, or some other situation specific to the ATE environment.

- After the cyclical redundancy check number has been compressed and stored, the verification procedure for testing in setup and in the production test environment is much faster.

7.2.4 The 'verify' test



The "verify" test compares the data on-chip directly with the data source file used by the "program" test to program the device. This test should not be used for production environments because it is time consuming. It should not be dependent on the "crc" test for program verification because the "verify" test uses the same data access mechanisms as the "program" test.

The "verify" test should be used on a known good board that has been programmed by mechanisms other than the HP 3070 ATE. Since the flash device has been programmed correctly, the "verify" test works as expected. because the data definitions are correct. If the test fails, the following problems may exist:

- The data is applied to the wrong data pins. For example, the high byte is switched with the low byte.
- The data source doesn't match the pre-programmed data.
- Control line offsets are not correct.
- Loads are not set properly.
- Threshold levels are incorrect.

  If this is the case, it is likely that the Flash Family power levels have been set incorrectly in the "verify" test. Compare the actual power levels to the levels specified in the data sheet to determine the correct threshold settings.

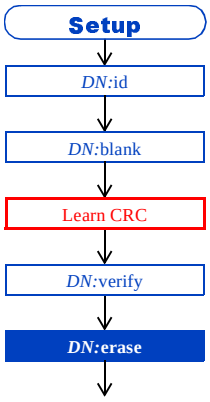
- All devices have not been disabled.

  If this is the case, run a working "blank" test on the device with pull-down loads. The test should pass. If not, upstream devices are driving the databus.

- The device type is not correct.

  If you are following the pre-programmed setup process described in **"The 'id' test"**, then this problem should have been discovered by the "id" test.

7.2.5 The 'erase' test



The "erase" test uses the flash device's automated erase algorithms to fully erase the memory device. Partial erasure is not typically done in the development phase of OBP setup. Problems with this test are not likely, especially if you have performed the "id", "blank", and "verify" tests successfully on the pre-programmed device. After the test passes, the device should read only FFs. You can verify this by executing the "blank" test.

If the "erase" test fails, possible problems follow:

- On Intel® devices, the erase algorithms return an activity complete response, and then a Full Status Check procedure is performed to verify whether the erase operation succeeded or failed. For example, a 28F200BX part uses the following algorithm to check for read failures:

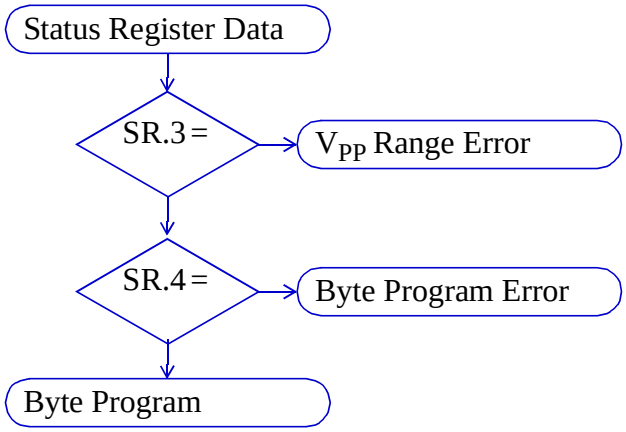
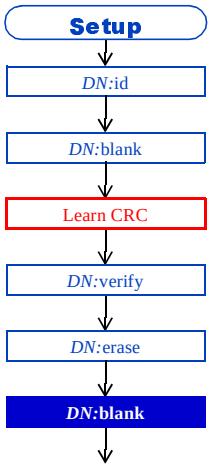


Figure 7-6 Intel® Full Status Check Procedure

If the register read failure occurs in the Full Status Check, then the device might require special programming voltages. In this case, ensure that the power supply voltages are set correctly on the testhead.

- If everything seems to be set correctly and the test still fails, ensure that upstream devices have been disabled.

7.2.6 Verifying an Erased Device With the ’blank’ test



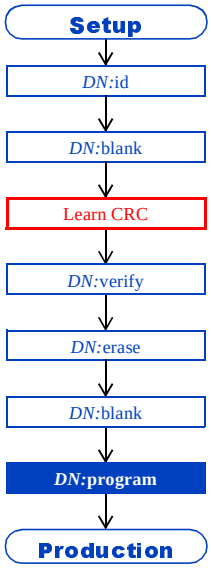
The "blank" test verifies flash device erasure. If the "erase" test passes, at this point, you will verify that the "blank" test is successfully reading the blank device. Execute the "blank" test to verify its operation.

Flash70

The "program" test does not take advantage of extra Flash70 features unless the Flash70 algorithm is enabled in the board config file.

See **Step 1: Configuring the board ’config’ file, on page 6-13**.

7.2.7 The ’program’ test



After you have verified that all tests in your flash OBP test suite are working, verify that the "program" test will write the correct data onto the flash device.

Since the purpose of this setup is to determine if the programming mechanism works with the data the board designers have provided, you need only to program a portion of the data onto the memory device. For this reason, the debug version of the "program" test programs the device without segments.

The verification process for the "program" test follows:

1. Ensure that the programming data source has been copied to the local "digital/" directory because the data source will be called by the test.

2. To verify that the "program" test works properly, run the "program" test for the flash device under test from a BT-Basic window. For example, enter **test "u2:program"** to run the test that programs the digital device named, "u2", in the board file.

3. If the "program" test passes**,** then verify that the data has been entered accurately. Display the data content in Pushbutton Debug, and compare it to the expected data. Or, run the **“The ’blank’ test”**.

For details on data verification techniques, read **Verifying that Programmed Data is Correct, on page 7-21**.

4. If the "program" test fails, determine the cause.

7.2.7.1 Troubleshooting a Failing "program" Test

If the abreviated "program" test fails, there are several possible causes. Look for the same type of errors that occur on standard digital tests. The "program" tests can fail for the following reasons:

- The library vector cycle time may be too fast for the fixture and board.
  - If using Flash70, don't set the receive delay longer than 100n or the dynamic vector timing will expand unnecessarily.
  - Signals are not reaching the device. Use *verify nodes* to verify access.
  - An automatic feature of the new compiler may not operate properly. Use "generate flash inhibit" statements to remove suspects one at a time. Then execute the "program" test in BT-BASIC to quickly observe any improvement.
- V_{PP} is not set at correct levels.

NOTE

This failure can be seen in Intel® devices by monitoring the Status Register. One of the pins will reveal the V_{PP} failure.

- Control line offsets are not set correctly.
- Threshold levels are incorrectly set in the Flash Family section of the "program" test.
- There are devices that have not been disabled upstream.
- The device type is not correct.

7.2.7.2 Expanding the Test to the Full Memory Size of the Device

When the test is working as expected, the repeat loop size may be expanded to program the full memory size of the device. Usually the size of the repeat loop is commented within the test. It is not necessary to adjust the repeat loop to the exact size of the data to be programmed, as the features that perform automatic FF stripping and extra segment removal will minimize the test appropriately. Simply expand the test to the full repeat size and uncomment the "segment" and "end segment" statements. After compilation, execute the test to ensure it will program the entire device properly.

If the test passes, evaluate the data with the "crc" test, if there is already a learned "crc" from a known good board. If the test passes, but there is no known good board, use the "verify" test to verify the data against the data file. Remember that the "verify" test should use the same data structure as the "program" test. Examine the data carefully using the "blank" test in the debug mode to check that the beginning, the end, and other address locations in the device are properly programmed.

If the device programs correctly, until the last data location of the device fails, you may need to add the end of data instructions to the test. For example, most devices will not respond as expected if a second write is attempted to a previously programmed data location. This problem arises when the test runs out of data before completion of the test. The default action is to apply FF at the highest address location for the remainder of the repeats. If the highest order address is already programmed, this action will result in a failure since the location cannot be programmed to FF. To use a location that has not been programmed, use the end of data command "reuse" to reprogram a location containing FF or "unused" to program a location not defined in the data file. For each of these actions the system selects the address to be programmed. In some cases, these addresses might also interfere with data programmed prior to this program operation. Possibly, serial numbers that have been loaded by another test could be the source of interference. In this case, you can select a specific address to be programmed with FF. This is the "user" end of data option.

One good tool for validating the "program" test is a *55, AA, 55* data source. This data source may be used as the source file. Then the blank check can be modified to verify alternating AA and 55. Since it checks every location, and is easy to use with debug, failing address locations can easily be identified. A contributed utility, genSrec, can be used to generate any size s-record with a variety of

data contents. Some of these files are also found in the "contrib" directory. To determine the content of a user file, the contributed tool, *flashDump,* can be used. This parses the data file as the compiler would. Along with the PDL generation tool, these contributed files are quite useful.

If the tests operate properly, the user has an opportunity to improve the operation of the test speed. First determine the true operating speed of the test by using a construct similar to the following in the BT-BASIC command line:

A=msec | test ’u3:program’ | print (msec -A)/1000; seconds

Notice that the test will take several runs to achieve full operation speed.

Changing the *vector cycle time* and adding *hardware waits* may improve the speed of the program, . Hardware waits can be used on newer flash devices that have separate output pins to indicate a busy state. Hardware waits may only be used if functional, combo, or flash70 testing is available on the system. If the device and system meets these criteria, try adding a hardware wait to the digital test. This does require a special "trigger" resource in the fixture.

The vector cycle may be increased up to 80n if the fixture and the device will operate at this speed. If flash70 is available, it is important to keep the receive delay 100n seconds or less for most efficient generation of the expanded dynamic vectors. Try to decrease the vector cycle. If intermittent failures occur, return to the standard vector cycle.

7.2.8 Obtaining Speed Improvements with Flash70

You can obtain the greatest speed improvement for flash tests by using Flash70 software. With Flash70, the new dynamic vectors utilize hybrid card and ControlXT card resources more efficiently and larger segment sizes can be used.

To optimize test speed, try multiplying the standard library segment size by 16. If the test will not compile, reduce the segment size in half. Continue this process until the test compiles. This results in the most efficient execution of the test. Because of the automatic FF stripping and segment removal features, there is no need to modify repeat loop size. Use the standard library repeat loops, so the test will automatically program only the actual data contained in the data file. Using a standard segment size limits the number of trials needed to

find the best segment number. You should not attempt to make minor adjustments in repeat loop and segment size. Time improvements will be minimal and the risk of error high.

7.2.9 Notes about Debug with Dynamic Vectors

Do not use debug on fully expanded tests, unless absolutely necessary. However, if it is necessary to evaluate a failure high in data memory, there is new listing features available. The compiler list option now generates the starting and ending vector for each segment. The compiler pads the end of the debugable version of the test with extra NOP or P vectors to avoid confusing displays at the end of segments. These extra vectors are not included in the compiler list information.

In the rare case when segmented tests require debugging, the graphics display may not always be accurate. The vector content will be as expected, but the count will be off by a number of vectors and the source code will not track properly. It is easier to find a recognizable address and data and count from that known point to evaluate the problem.

7.2.10 Notes About Debugging With Flash70.

If errors are encountered with the "program" test, debugging may be required. Generally, it is easier and most effective to debug non-segmented tests. The multiple vectors that make up the expanded dynamic vector can readily be seen in debug. The order of the vector build up is determined by the order of the drive statements in the execution statement. The control lines however will always be asserted in the last vector.

Figure 7-7 Debug Display of a Flash70 Test

The two diagrams show a normal test and a flash70 test. Observe the 4 vectors that make up the dynamic vector on the flash70 test. First the high order address pins change, second the lower 9 address pins change, followed by the high order data, and finally the low order data and control lines (not shown).

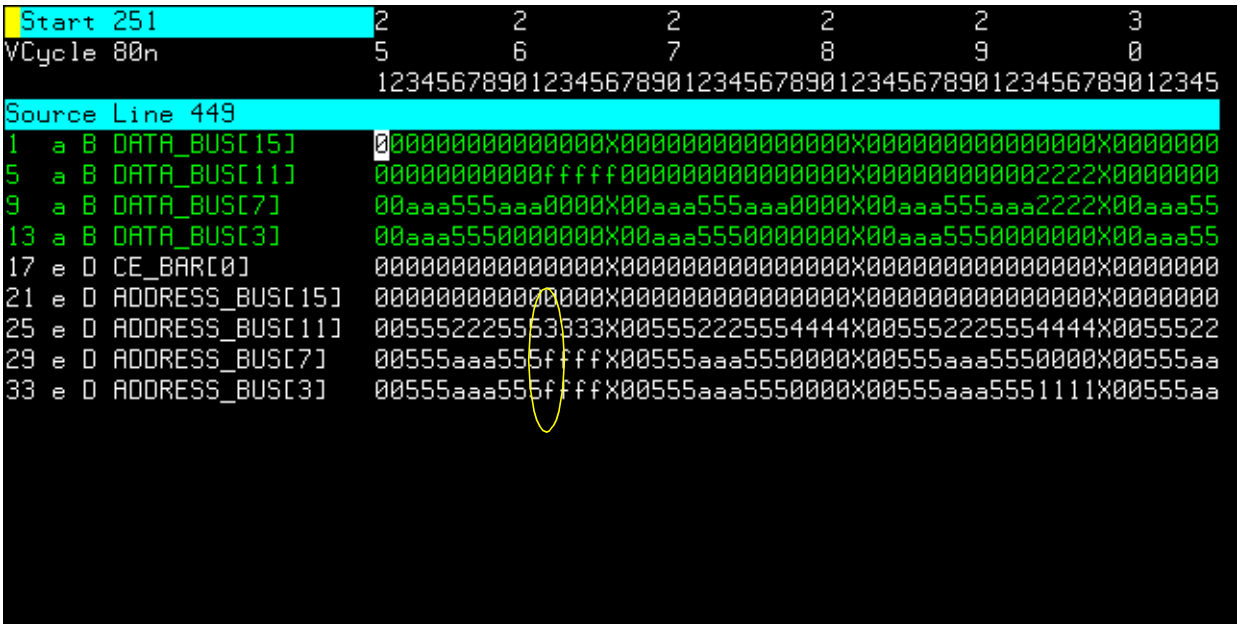


Figure 7-8 Debug Display of a Standard Flash Test

Debug can be a useful tool for Flash70 tests, if you are aware of some anomolies. The debugger was developed for standard digital tests. With the special timing of the dynamic vectors, modifications to timing within debug are of limited value. If the vector cycle is changed, all vectors, including those making up the dynamic vector will be changed. Some of the timing within the dynamic vector will adjust to an indeterminate state. If the timing is changed, the only way to return to the original state is by updating the debug session. Modifications of timing do have some limited value, but they need to be followed up by direct modifications and recompilation of the original test. If debug modifications of the vector cycle or receive delay result in a working test, there are two possibilities. Perhaps the vector cycle of the ordinary vectors is too short or perhaps the dynamic vectors do not create a valid data cycle. Try modifying the vector cycle in the test and recompiling. Re-execute the test. If the test continues to fail, try turning off the dynamic vectors. Recompile and execute. This process may help you find the cause of the failure.

The new flash compiler has many useful features. However, not all of these features are supported in debug. For example, when debug is attempted within a segmented test and the test runs out of data, it will use the end of data setting to program until a segment is completed. Any unnecessary segments are not executed in the BT-BASIC execution statement. However, in the debugger, all segments are executed even when no data is available. Do not expect segmented tests to stop segment execution once the end of data is reached. All other aspects of the test execute as expected. The end of data address continues to be executed until all repeats are completed. When the test is working properly, recompile the final test without the debug option. This results in the fastest possible test. For instance, Throughput Multiplier tests will not operate in parallel when compiled in debug

7.2.10.1 Verifying that Programmed Data is Correct

After all the tests in the OBP test suite have passed, you might think the data is programmed correctly on the flash device. Usually, this is the case. There are instances, however, when the OBP test suites work properly but the data is still programmed incorrectly. For this reason, it is best to conduct a more extensive validation of the programmed data manually after the "program" test has passed. This is particularly important on 16-bit data bus devices or multiple device topologies.

If the test setup process for pre-programmed flash devices outlined in this chapter have been followed, you know the tests seem to work

correctly. Therefore, data application from the file should also be correct. Use a known good board to quickly validate this assumption. The procedure is shown below.

1. Use the "blank" test in Pushbutton Debug to read the data for the device on the KGB as described in **Displaying Accurate Addresses With the New Display Group, on page 7-10**.

2. Replace the board with a board programmed via "program" and display the data again. This requires execution of the testplan to return to a powered up state, but the hexadecimal display will be retained.

3. Compare the data programmed with the "program" test to the known-good result.

   - ♦ If the data is identical, the OBP test suite has been set up correctly and is ready for production.

   - ♦ If the data is not the same and the individual tests in the OBP test suite have passed, there are a few likely scenarios.

     - The data bits have been reversed (see **Correcting Reversed Data Bits, on page 7-22**).

     - Address locations are skipped. Use the step procedure to correct this problem.

7.2.10.2 Correcting Reversed Data Bits

In the case of a 16-bit data bus, high order and low order data bits are often reversed. This is due to the little Endian vs. big Endian use of standard Motorola® S-records or Intel ®hex records. This happens because the data source records do not define the most significant byte orientation of data. You can remedy the problem by doing the following:

Problem: Data Is Reversed:

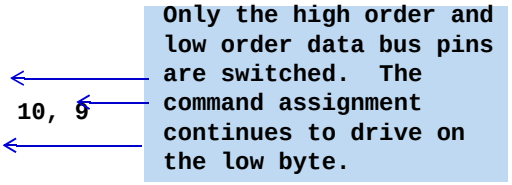
```
Address 0000 Data    1100                      0011
                     3322    Should Be  >>>>   2233
                     5544                      4455
```

Solution: Assign high order bits to low order:

```
Existing assign statement:
      assign Data_bus to pins 16, 15, 14, 13, 12, 11, 10, 9
```

```
      assign Data_bus to pins 8, 7, 6, 5, 4, 3, 2, 1
      assign Data_cmd to pins 8, 7, 6, 5, 4, 3, 2, 1

Change to:
      assign Data_bus to pins 8, 7, 6, 5, 4, 3, 2, 1
      assign Data_bus to pins 16, 15, 14, 13, 12, 11, 10, 9
      assign Data_cmd to pins 8, 7, 6, 5, 4, 3, 2, 1
```



Modifying the pin assignments in the "program" test as shown above sets your test up to match the intention of the board designer.

7.2.10.3 Address Misalignment.

Another situation resulting from a 16-bit data bus is address misalignment. This is documented in the B.02.50 on-line users documentation. To correct these errors, use the "step" command and create dummy address pins.

7.2.10.4 Data Addressing for Data Records Larger Than 8 bits

Data is typically provided in 8 bit addressing formats. Recently, however, board designers have provided some test developers with Motorola® S-Record and Intel® Hex Record formats that use other than 8 bit data addressing. This variance is not part of the formal specification for either record type and results in incorrect data programming if the variance is not specified in the appropriate flash tests in the **OBP** test suite.

The error in programming occurs, because the standard interpretation for reading data records assigns one address per 8 bit data chunk. If, for example, the data file contains 16 bit addressing, then each 16-bit chunk of data would be incorrectly assigned one address. This standard interpretation results in records being mapped on top of each other, when reading 16 bit records.

An example of this situation follows:

8 bit addressing

```
:202000000A08B00BC2AAE110028AEF00029AE000F05AE800046BD80BFF9006990690C63IFC9
:202020000080BF70006990690C588080BF60006990690C58807DAE00007D0C5B807DAE8000AC
:20204000B00C5A807DAED9007D0C6380898B00BC21AE010004AE000042BE7DAEBD277D0FE0
:20206000008AAE000026AE1000075EE807075DA80000B9639064906A9000B100B9C4BEFF7F6C
:20208000080A780BF000060B11FBBA09009BF0001C4BEFF03A09009BF0008C4BEFFF7A09059
:2020A0000000BF60006990690C588080BFFFFF07BB3CBE80BF1F12C0BF0080108809BF001083
```

16 bit addressing

```
:202000008AA080A0BC2AAE110028AEF00029AE000205AE800046BD80BFF9006990690F6380
:20201000B9008B8E806990690C588080BF60006990690C58807DAE00007D0C5B807DAE8000
:20202007A80205807DAED9007D0C6380898B00BC21AE010004AE000042BE7DAEBD277D0FE
:202030090808B0026AE1000075EE807075DA80000B9639064906A9000B100B9C4BEFF7F6C
:20204000BFB0FFBF000060B11FBBA09009BF0001C4BEFF03A09009BF0008C4BEFFF7A09059
:202050002051BF9FF990690C588080BFFFFF07BB3CBE80BF1F12C0BF0080108809BF001083
```

Notice that the 16 bit addresses increments by 10. Since 20 bytes of data per line are programmed, every other line of data is overwritten if the "16 bit" data modifier switch is not added to the file command.

Figure 7-9 8 bit versus 16 bit addressing

The HP 3070 software compensates for the variance in addressing schemes by allowing test developers to specify which addressing scheme their data records use. In the example above, adding the data modifier "16 bit" to the "file" statement in the "program" test for your flash device fixes the addressing problem.

NOTE

Since this addressing variance is not standard for these formats, VCL cannot accurately determine whether the data records are 8 or 16 bits. You can supply the correct information to the software by using data modifiers.

7.2.10.5 Addressing Data Modifiers

The file statement within a **VCL** data block must be modified to allow specification of how many bits will be allocated per address. The modification will be made to the data specification of the file statement. Examples for 8, 16 and 32 bit interpretations of a Motorola S-record file are shown below.

```
file "data" 131072 S-record data
file "data" 131072 S-record data 16 bit
file "data" 131072 S-record data 32 bit
```

Note that the default and the 8-bit are identical:

```
file "data" 131072 S-record data
file "data" 131072 S-record data 8 bit
```

The use of the data modifier will change how the system interprets the addresses in the Motorola S-Record or Intel Hex Record. A 16-bit modifier specifies that the address will be incremented once per 16 bits rather than once per 8 bits before other modifiers such as the step translation be applied.

IMPORTANT

Keep in mind that the "8 bit", "16 bit" and "32 bit" modifiers are NOT specifying the width of the data bus of the device being programmed. The modifiers specify the format used in the Motorola® S-Record or Intel® Hex Record data files only.

Chapter 8
Series and Parallel Programming

Rev. A

There are several topologies that can be used to improve your flash programming speed. Some common topologies and programming strategies are described in this chapter.

This chapter requires an understanding of the following:

- Flash programming fundamentals. See **Chapter 1, "Introduction to Flash Programming."**
- What makes a board DFOBP compliant. See **Chapter 2, "Design For On-board Programming."**
- How to generate flash digital libraries for the board under test. See **Chapter 5, "VCL Syntax for Flash OBP."**
- How to setup flash OBP test suite for production. See **Chapter 6, "Generating Flash Digital Tests."**

This chapter describes:

- Flash series programming in cluster tests. See **Series Flash Topology Cluster Test Programming Model, on page 8-2**.
- Flash parallel programming in cluster tests. See **Parallel Topology Cluster Test Programming Model, on page 8-5**.

8.1 Series Flash Topology Cluster Test Programming Model

Although you can use two tests to program two devices on the same bus sequentially, it is easy to program two or more devices with one test. This test strategy sends all the programming commands to both devices at the same time. The dynamic vectors, which give each device the appropriate data, select one device. The first device to receive the programming data will be the first device to be polled if data polling is required. For best results, both source files should contain contiguous address locations. This way, only one address is required for programming both devices.

The procedures for performing serial programming for a flash culster topology follow:

1. Select one of the working program tests in the digital subdirectory to be the multi-device programming test.

2. Add the nodes that apply only to the other device(s) to the assign statements. Most of the pins will have common nodes in this topology, so you should not need to add many.

```
assign CE_bar  to pins 10     !< original test

assign CE_bar_u9  to nodes “U9_READ”

input CE_bar, CE_bar_u9
```

3. Add the node to each vector that sets that pin type

    a. .Set it to the same value for every vector, except for the dynamic host vector. For this, keep the original vector intact.

    b. Duplicate the vector for the new device.

In some cases, the standard tests keep CE asserted at all times. For these tests, you may need to make minor changes. For example, prior to the dynamic vector, you may need to add a new End_cycle_full statement to take the CE_bar high for both devices.

The following example shows how to change the dynamic vector.

Original vector:

```
vector  Data_W
   initialize        to Keep_Control
   drive Data_bus
   set   WE_bar      to "0"
   set   Data_bus    to "0000"
end vector
```

New vectors:

```
vector  Data_W_u8
     initialize        to Keep_Control
   drive Data_bus
    set   CE_bar      to "0"   !<<< added set statement
   set   WE_bar      to "0"
   set   Data_bus    to "0000"
end vector

vector  Data_W_u9
     initialize        to Keep_Control
     drive Data_bus
     set   CE_bar_u9      to "0"  !<<< added set statement
     set   WE_bar      to "0"
     set   Data_bus    to "0000"
end vector
```

The **wait** or **homingloop** vector requires similar treatment. The homingloop for data contents requires separate vectors. For the **wait** or **homingloop** on devices with ready pins, both devices can be activated with one vector.

A new data block is required to reference the data file for *"u9"*. If contiguous address data files are in use, only one data block is required for the address, since both files have full addressing. Using contiguous addressing results in faster test speed. This example below this scenario. Since in this topology, the buses are the same, groups Data_bus are used in both data blocks.

Original data blocks:

```
data  Data     to groups   Data_bus
   file  "2Mx8.data" 1048576 s record data
end data

data  Address  to groups   Address_bus
   file  "2Mx8.data" 1048576 s record address
end data
```

Add one new data block:

```
data  Data_u9     to groups   Data_bus
   file  "2Mx8.data_u9" 1048576 s record data
end data
```

The execution statements should remain the same, since they have been enhanced to add selects for the new device or devices. The only changes should be as follows:

```
...
execute End_Cycle
execute End_Cycle_full  ! take CE high for both parts
execute Data_W_u8  drive data Address  drive data Data
execute End_Cycle_full
execute Data_W_u9 drive data Data_u9  ! write to second device

execute End_Cycle_full

 wait 4u
 homingloop  60000 times
    execute XSR_Ready exit if pass
    execute Three_state
 end homingloop


  homingloop  60000 times      ! wait for second device
    execute XSR_Ready_u9 exit if pass
    execute Three_state
  end homingloop

 next Address            !point to next address for both
 next Data               !point to next data for u8
 next Data_u9            !point to next data for u9
```

After the repeat loop, add a separate read status register to ensure each chip has valid programming.

8.2 Parallel Topology Cluster Test Programming Model

The HP 3070 can provide any width of data for dynamic vectors. This means, if identical device types reside in parallel on one bus, as in **Figure 2.3.3.2**, they can all be programmed with one test. The test time required to program all four devices is equivalent to the time it would take to program one device. The decision to use a parallel programming model must be made prior to program development because this model utilizes many more resources than a single test would. Adopting this model can yield significant improvements in programming time.

This test model creates a cluster test that expands the library to the full width of the data bus. The test sends the same commands to each device, but sends different data to each device based on the data block assignments.

8.2.1 Performing a Parallel Test

Create a new PDL with all the uniquely noded pins of the flash devices included.

1. Create appropriate setup tests with definition sections as described below, for the PDL.

2. In the board file, add a new device that contains enough pins to contain all the independently noded pins. For example, u8_u11. This is easy to do by copying parts of one device into the new device.

3. In Board Consultant, save the board with the "Board File List Format" set to "Device" in the Global Options Form. This creates a board file with device references only, so you can easily copy parts of the four existing devices to the new devices, u8_u11. With these additions, you can proceed with board development. You can make changes to the test while the fixture is being built.

4. Modify the test as described below.

Definition Section:

a. Add the required data buses for the additional devices.

b. Add any control lines that are not on common nodes on the board.

c. Assign "dynamic" to any new data buses.

d. If you'll be using different data files for each device, the data bus and data block will need to be assigned independently.

If you have a single data file containing, in this case 64 bits of data to be distributed to the devices, you'll use a single data bus assignment and data block. Careful analysis of the data structure is required to make this determination. Each case cannot be described here, so this is an exercise for an experienced user.

8.2.2 Add New Data Blocks to Reference Separate Files for Each Device

```
data  Data_u8     to groups   Data_bus_u8
   file  "2Mx8.data" 1048576 s record data
end data

data  Address  to groups   Address_bus
   file  "2Mx8.data" 1048576 s record address
end data
```

Add new data blocks for each new bus:

```
data  Data_u11     to groups   Data_bus_u11
   file  "2Mx8.data_u9" 1048576 s record data
end data

data  Data_     to groups   Data_bus_u10
   file  "2Mx8.data_u10" 1048576 s record data
end data

data  Data     to groups   Data_bus_u10
   file  "2Mx8.data_u11" 1048576 s record data
end data
```

Vector Section:

Modify each vector by adding the additional data bus with matching data content to the original vector.

NOTE 

Be sure to include the assertion of the control lines.

Original vector:

```
vector  Data_FF
   initialize        to Keep_Control
   drive Data_bus
   set   WE_bar      to "0"
   set   Data_bus    to "FFFF"
end vector
```

New vector:

```
vector  Data_FF
  initialize        to Keep_Control
   drive Data_bus_u8, Data_bus_u9, Data_bus_u10, Data_bus_u11
   set   WE_bar      to "0000" <<< WE_bar is assigned to all
WE pins
  set   Data_bus_u8    to "FFFF"
  set   Data_bus_u9    to "FFFF"
  set   Data_bus_u10   to "FFFF"
  set   Data_bus_u11   to "FFFF"
end vector
```

After correcting the vectors, additions to the executable section should be minimal. Change the dynamic vector to drive all buses.

The new executable section should look like this:

```
segment 2048
   repeat 1048576 times
   execute  Setup_Program_Cmd     ! data = 40
                                  !!each command drives the correct data to all 4 devices.
   execute End_Cycle
   execute    Data_W drive data Address drive data Data_u8
              drive data Data_u9 drive data Data_u10 drive data
              Data_u11

         execute End_Cycle

         wait 4u
         homingloop  60000 times
                               !!  The XSR vector has been modified to look for each device
           execute XSR_Ready exit if pass
           execute Three_state
         end homingloop

            next Address       !point to next address for any device
         next Data_u8       !point to next  data word
         next Data_u9       !for each data block
ADVICE   next Data_u10      !and each device
         next Data_u11
   end repeat
end segment
```

Hardware waits can be used if there are less than three conditional pins in the cluster, since three hardware trigger resources are available for a single digital test.

This general example should help you to generate this test. If you created the setup tests prior to board development, you will have plenty of time to develop the test because the resources are contained in the fixture file.

You can use the standard libraries for production until you've validated the new tests.

HEWLETT PACKARD

Appendix A

Online Guide to Acronyms for Flash OBP

Online Acronyms	
blank	Any flash device that has been erased. It is also the suffix of the test library that confirms all memory cells of a flash device read FF. See **Table I on page 6-5**.
ATE	Automatic Test Equipment validates PCB construction by seeking assembly faults, open traces, shorted traces, misaligned devices, and missing integrated circuits (ICs). Design engineers use ATE to perform in-circuit testing on assembled printed circuit boards (PCBs). The HP 3070 ATE can be used to program your flash devices in-circuit.
BSDL	Boundary Scan Description Language. As used in this document, BSDL implies devices that follow the IEEE STD. 1149.1 conformance document.
CFI	Common Flash Interface. This is the flash standard the industry is moving towards.
JTAG	Under Construction.
DUT	Device Under Test.
CUI	This is the Command User Interface embedded algorithm that is written on the first 2 bits of the flash memory boot record. The CUI controls the writing of data onto the flash device by toggling CE# and OE# pins as appropriate.
OE #	Output Enable control line on a flash device.
CE #	Chip Enable control line on a flash device.
WE #	Write Enable control line on a flash device.
CRC	The suffix of the test library that performs a Cyclical Redundancy Check on a flash device and then compares it to a learned result. See **Table I on page 6-5**.
erase	The suffix of the test library that erases flash devices to prepare them for programming. See **Table I on page 6-5**.
FPGA	Field Programmable Array.
IC	Integrated Circuit
ICT	In-circuit Test
id	The suffix of the test library that uses embedded algorithms to identify a flash device. See **Table I on page 6-5**.
IPG	Integrated Program Generator subroutine that is invoked in BTBasic or Test Consultant.

Online Acronyms	
IPGTC	Integrated Program Generator Test Consultant subroutine that compiles test files into executable objects.
JIT	Just In Time manufacturing.
KGB	Known Good Board.
Mb	Megabit (1,048,576 bits).
MB	Megabyte (1,048,576 bytes).
OBP	On-board Programming
PCB	Printed Circuit Board.
PDL	PDL is a language used to describe devices. In some HP 3070 manuals, PDL represents both Parts Description Language and Part Description Library. In the Flash Online Guide, the PDL acronym is used when describing the file that points to the flash programming tests such as "am29f040_program".
program	The suffix of the test library that programs flash devices with formatted hex data. See **Table I on page 6-5**.
SOP	Small Outline Packages.
TM	Under Construction.
V_{CC}	Programming voltage required to implement flash.
VCL	Vector Control Language.
verify	The suffix of the test library that compares data on a flash device with a formatted data file. See **Table I on page 6-5**.
V_{PP}	Programming voltage required to implement flash.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Symbols

Numerics

A

B

C

D

E

F

H

I

M

N

O

P

R

S

T

U

V

W