Agilent Technologies

Writing Flash Memory

Flash memory is traditionally programmed on PROM programmers, but some manufacturers report a one percent damage rate due to the extra handling steps required! With technology improvements, smaller packaging size encourages reduced handling of flash memory. Programming the flash after it is installed on the board using automatic test equipment (ATE) is the best solution. ATE programming provides many other benefits including Just-In-Time manufacturing and simplified inventory control. The flash memory manufacturers can provide information about the benefits and limitations of on-board programming (OBP). Short of self-loading of the flash within the application, ATE programming is the best choice for flash.

The Agilent 3070 family of board test systems makes programming flash memory a snap! Standard hardware and software makes flash programming easy and straightforward. It requires no system upgrades. Agilent supplies flash libraries, but creating tests for newly released devices is quick and easy. The Agilent 3070 digital subsystem handles industry standard data files with ease, making engineering changes simple to implement, and Just-In-Time manufacturing automatic. Flash programming is one more tool in Agilent's extensive ATE toolbox.

Ease Of Programming

Programming flash with an Agilent 3070 system requires no added hardware or software. The Agilent 3070 family uses its built-in digital architecture, so there is no impact on system node count. Standard digital test capabilities make programming flash memory easy. Since standard Vector Control Language (VCL) is used, test engineers can quickly write tests to program flash if the devices are not already in the Agilent 3070 library. When a board is designed with ATE programming of flash in mind, the Agilent 3070 family provides easy, cost effective loading of any kind of flash memory.

Writing a flash test is easy using VCL. The language has a few command features specifically designed for data loads into any digital device test. Here's a look at these commands:

Flash Memory Commands

block	defines data block declaration segment
file	names file containing Intel Hex Format or Motorola S-Records
next Data	increments data from the file
next Address	increments address from the file
generate static test	freezes data from data file into compiled code
dynamic	identifies address and data buses for efficient memory utilization

VCL algorithmically handles the data format, so no user interaction is required. An Intel Hex Format, Motorola S-Record, or decimal data file provides the addresses and data. This data file can contain the full memory contents or noncontiguous blocks. Either way, the addresses and data are correctly retrieved. This clean, standard format interpreter has several advantages. The device can be programmed in blocks simply by changing the repeat count. If the code is changed frequently, or for version control, only the data record needs to be changed. No recompile is required.

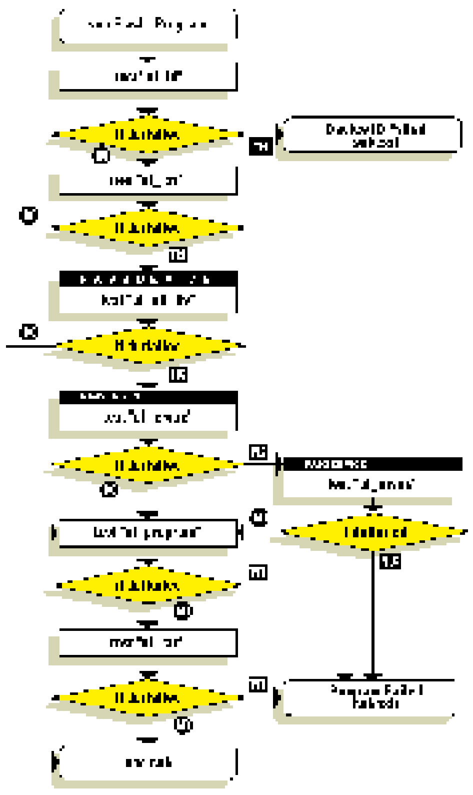
Flash programming requires more than a single programming test for proper use in a manufacturing line. Several different tests are required to identify, erase, program, and verify the device. These tests are each simple variations on a theme. The identification test simply reads appropriate register locations to verify the manufacturers ID code and the device ID. Erasure of the memory may require writing commands to an address in the block to be erased. Verification of erase (blank check) reads FFh at every location. Verification of programming uses a similar read of all locations with a compress option, which creates a cyclic redundancy check (CRC, see below). The programming of the device generally requires that a program command be written to the device followed by the address and data to be programmed. As the device is programmed, the test waits, polling for completion status.

Test Management for Flash Programming

With newly manufactured boards, the flash devices arrive blank at the in-circuit test stage, so simple programming is all that is required. In a real production environment some boards may be returning from repair, or returning for an upgrade with new data. For these devices, the test system needs to verify erase and programmed status. In some cases, boards are rejected for device failures, such as wrong device ID, or devices that won't program. The various tests required to handle these contingencies are cleanly controlled by a testplan written in Agilent 3070 BT-BASIC protocol.

The test engineer adds the BT-BASIC commands that control the decision tree for programming devices to the existing testplan, as shown in the flow diagram. When verify erase and verify CRC are used for evaluation rather than failure mechanisms, the board should not be rejected. Execution of "pass device" clears the DUT failed status before proceeding. Immediately clearing dutfailed in this way, also retains the boardfailed status. By carefully following the recommended steps, redundant operations are avoided and test time is saved.

The flow chart diagrams the programming process, but one technical issue remains: Some flash memory requires a 12 volt supply to program or erase memory. Ideally, the 12 volt supply will come from the board itself. Some boards do not allow for on-board programming, so the Agilent 3070 system supplies the programming voltage. The testplan will set the power supply to 12 volts when it is needed. General purpose relays provide additional power supply options in special cases.



For some manufacturers, version control is required for customizing boards. A different version of the digital test for programming and CRC check is used depending on the version of the board being produced. The testplan must query the user to learn the version of code to be programmed. Then it must respond by calling the correct digital file. Currently, the test engineer adds this to the testplan. Effortless version control is planned for future software enhancements.

The Agilent 3070 System Readily Solves Your Flash Programming Dilemma

If fast, easy flash implementation using industry standard data files for easy engineering changes is important, the functionality built into every Agilent 3070 test system will deliver.

Example of Programming Flash Memory

When a new flash device becomes available, you may have it on your engineering prototypes before your ATE vendor has it in their library. When this happens, as it happened to me when an engineering sample of the Intel 28F002bc, 2-Mbit Boot Block Memory arrived on my desk, the Agilent 3070 makes creating the required tests easy. Within an hour of putting the device on the tester, I could program and read the device. A day later, I had a complete test suite. Here's how to accomplish this on a new device.

First, create a setup file for fixture and test generation. Using the word "dynamic" in the assign statements for address and data buses, signals the test system to assign pins efficiently. This way pin resources are selected so that dynamically changing pins, such as data, are placed in blocks of pin RAM memory, speeding memory loads. The assignment must be in the digital test during fixture development to produce this benefit.

The best way to create a working test environment is by the use of a part description library. This library, named as the device part number, contains the various tests required for flash. When this part description library is used, multiple executable tests are generated to erase, crc, blank_check and program. Create a part description library, 28f002bc, which cross-references each test to the pins on the part. Copy the setup test to each library name referenced in the part description library. Run the full test development process on the board. The resulting fixture will support all tests. The tests created will be u1:erase, u1:crc, u1:blank_check and u1:program. When the fixture is complete, you can easily see where to begin working on development of the working tests.

A quick review of the data sheet clarifies the basic control of the 28F002bc to be command and register based. Examining the waveforms shows how to access the device. First write a VCL program to read the data on the device. This verifies the proper bus cycle sequence expected by the device. This is the basis for the verify, and display tests. Using receives of FF will verify if the device is blank. Adding an upcounter on the address bus, and a repeat loop of appropriate length, in this case 262144, completes the test. I like to compile this for debug, so I can examine the data content after programming or erasing the device. A copy of this test will later be used for the CRC test.

```
unit "Verify all FFh on flash"
   execute Init
   execute CE_low
   preset counter Address_count
repeat 262144 times
   execute Read_data_FF
   execute End_cycle
   count Address_count
end repeat
   execute End_cycle
end unit
```

The verify_id test is slightly more complicated, as it must access registers. There are two methods to read the identification of the device. I picked the command method, as it doesn't require a special voltage. To read the identification, first write the Intelligent Identifier command, 90, to the device. Any address will do. Now reading address 0 gives the manufacturer's ID, 89. Reading address 1 gives the device ID, 7C. This test will be used to ensure the proper device is installed on the board.

The next step is erasing the device. Flash memory can only be programmed to zero. For this reason, it must always be erased before programming, if you expect to get the correct data programmed. Otherwise, programming 66, at a location already containing 33, will result in 22. For programming or erasing the device, VPP and RP must be set to 12 volts. Now is the time to make the power supply adjustments on the Agilent 3070.

Memory Map

Memory Content	1 1 1 1	1 1 1 1
Write	0 0 1 1	0 0 1 1
Changes	| |	| |
Memory Content	0 0 1 1	0 0 1 1

Write	0 1 1 0	0 1 1 0	WSM	complete OK
Changes	|	|	Changes Program	complete OK
Memory Content	0 0 1 0	0 0 1 0	Contents	WRONG!

During the first two steps of test development, the read sequence, write sequence and command structure of the device was revealed. Now erasing is a matter of determining the proper steps. Examination of the erase sequence shows that writing the erase commands to any address within the block erases that block. The two command sequence is written to block 1, at address 0. The erase is set up with data byte 20 and then confirmed with data byte D0. To determine when the Write State Machine (WSM) is finished erasing, the status register is polled. Execute a homingloop reading the device. Exit the loop when the status register bit, D7 goes high. Be sure the homingloop is long enough to ensure proper erase time, .3 seconds for parameter blocks, .6 for the boot block. After the WSM is finished, it is important to ensure that the program was properly completed. Read the status register again checking for bits 3, 4, 5 to be 0. Failure of this read shows that the power supply is wrong, or the device couldn't erase. Clear the status register by writing 50. Repeat this for each of the 5 blocks. When erasure is complete, write FF to put the device back into normal read mode.

Now all the tests required to manage a production test environment are complete, except for programming. Development of the programming test is best done in small steps. First write a test to program one byte. This test is slightly more complicated than the others, One program command, 40, starts the sequence. Write the command to the address to be programmed. OK, where does this address come from? This is where the few special variable handling commands come into use. The file is defined just before the execution units. A segment defined as data assigns the Address and Data buses to a 262k Intel Hex record called I_hex. This is all that needs to be done to define the data source. Define the complete size of the hex record, because the program only updates the data with "next Address", or "next Data". You can program just one byte for initial debug despite the size being defined much larger. After the program is compiled, you can change the data simply by changing the contents of the hex data file, unless you are using "generate static test", a VCL command that "hard codes" the Intel Hex data into the program.

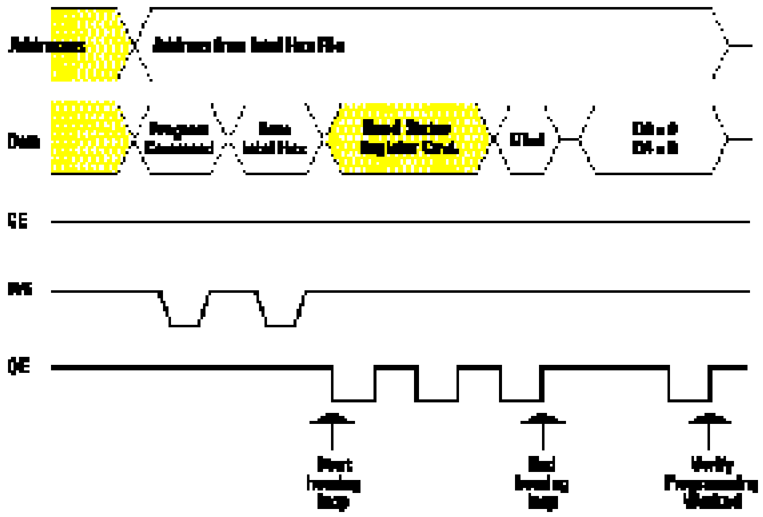
```
data Data to groups Data
   file "I_hex" 262144 hex record data
end data

data Address to groups Address
   file "I_hex" 262144 hex record data
end data
```

Start the programming sequence by driving the address on the bus. (See programming example.) This is accomplished by adding "drive data Address" to the end of a preliminary vector. The next vector would drive the program command 40 on the data bus, keeping the address. Next write the data to be programmed to that same address by adding "drive data Data" to the end of a write vector. Note that all vectors in the write portion of the test keep the Address and Data, or this staged process wouldn't work.

After the write, release all pins in the End_drive_data vector. This will eliminate backdriving of input pins, helping to protect upstream devices. For this device, a wait of several microseconds is required. Determine when the WSM has completed its work by using the status register read homingloop from the erase program.

When one byte is successfully being programmed, try 100 bytes by adding a repeat loop around the working write vectors. To increment the Address and Data, put "next Address" and "next Data" at the end of the sequence, before the "end repeat". After the repeat, add a full status check, verifying that the status register bits 3 and 4 are 0. Following this, clear the status register, and reset the device.

```
unit "Program 512 bytes"
   execute Init
   execute CE_low
   execute Clear_status_register
   execute End_cycle
repeat 512 times
   execute End_cycle drive data Address
   execute Program_cmd
   execute End_cycle
   execute Write_data drive data Data
   execute End_cycle
   execute End_drive_data
   wait 5 u
homing loop 40 times
   execute Check_WSM_status exit if pass
   execute End_cycle
end homing loop
   execute End_cycle
      next Address
      next Data
end repeat
   execute Check_program_status
   execute End_cycle
   execute Reset_device
end unit
```



Take time now to verify as many ways as possible that the programming is correct for all 100 bytes. It is easier at this stage than when programming the full memory range. Use your imagination on this, because it's important to ensure proper programming for production. The best way is to do a CRC learn on a device programmed by another method. This ensures that an artifact in your programming sequence isn't recreated in your read sequence.

After you are confident the test is programming properly, increase the repeat count to the full memory size and enclose the repeat loop in a segment. The segment size will need to be determined by experiment. For the 28F002bc, it is 2048. Compile this test without debug, because you do not want to debug such a large test. Also, you know it works from your earlier experiments.

The only remaining test is the CRC test. Use your verify all FFs test. Add compress statements to the data receive vector. Find a known good board. Turn learn on. Execute the test. Turn learn off. Now you have a CRC test that matches your known good board data. Try this test on a flash programmed by your test, and you'll know if you are successful.

There you have it, - how to create the test suite for a totally new part. It literally took longer to write this article, than to write and debug the programming test.

Good luck on your next flash!

by Julie Keahey, Application Engineer, Agilent Technologies