Software Overview 2

Introduction

The architecture of the FLASH 750 software is based on user interfaces that are oriented toward the DUT rather than the tester. The FLASH 750 system software is designed around the expected use of the test system by the end user: it is oriented toward *device test* rather than the *test instrument* or tester.

Key Concepts and Terms

Device-Oriented User Interface

The user interfaces of the FLASH 750 are centered around the device, rather than the FLASH 750 test instruments. This approach matches how test and product engineers think about a device.

Tester Instruments

FLASH 750 instruments include:

- High-voltage measurement unit
- Board PMU
- Per-pin PMU
- Per site Flash Algorithmic Pattern Generator (APG)
- Pin channel hardware
- DPS

Testing Flash Devices

FLASH 750 software supports four types of tests:

- DC
- AC
- Functional
- Characterization

DC Testing

DC tests detect gross device faults and measure power.

DC Parametrics

The DC operating parameters are checked for continuity, input and output leakage, output current, and device input current (I_{CC}). Faults are usually caused by manufacturing defects and process variations. These tests measure the current sourced or sinked and the voltage levels on the pins:

- *Continuity* ensures good contact between the tester channels and the device. Continuity testing is very important at the wafer probe level.
- S*horts*, *opens*, and *diode functionality* verify the integrity of the device packaging.
- *Leakage* on each input pin is measured by sourcing 0 volts on all input pins. VCC is then sourced on each input pin, and current is measured on the other pins in parallel by using the per-pin PMU (PPMU). Pass/fail limits are derived from the device data. All output pins are measured this way.
- *Ganged leakage* is similar to leakage tests except that groups of pins are connected together when the current is measured.

ICC

ICC measures the current drawn by the device from the device power supplies while the device is static or dynamic.

IDDQ

IDDQ measures the device current leakage. Typical IDDQ is very low; thus, unexpectedly high current measurements can help you locate manufacturing faults. These faults are usually partial shorts within the device, which cannot be detected during scanning.

AC Testing

These tests verify that the device meets or exceeds its AC specifications, which includes maximum frequency, setup-and-hold times, access times, propagation delays, and voltage levels representing logical values. These tests are often combined with functional tests. Faults are usually caused by manufacturing defects and process variations.

Setup and Hold Times

Devices have setup and hold specifications, which are the times a device input must be stable before and after the input is recognized correctly.

Functional Testing

Functional testing has three parts:

- Writing to the memory under test
- Reading back the contents of the memory
- Comparing them to determine correct functioning

These tests verify that the memory cells of the device are functioning properly by stimulating them with logical values (1's and 0's) and by verifying that the device responds with the expected values. A device that passes functional tests at the manufacturer's specified operating speed is 100 percent operational.Typical tests include speed sort functional and nominal functional.

Characterization Tests

Characterization tests use standard test techniques, usually functional testing, to determine the threshold values of the test parameters—the values at which a passing device begins failing. Normally these test techniques return a *go/no go* result. To determine the device characterization, individual tests vary test parameters as a test is repeatedly executed to determine the values at which the device fails. Characterization testing includes DC and functional testing with special characterization tester tools, such as datalogging, bitmap, schmoo, and search.

Production Tests

Production testing verifies the device operates within an acceptable performance range without determining the actual operating range. Production tests are optimized for minimum test times. A variation of production testing is incoming inspection testing, in which characterization and production tests are done on a sample or the entire lot.

Parallel Testing

The FLASH 750 system software manages the shared resources for the devices tested in parallel. Each device tested in parallel may not always be assigned an independent set of tester hardware. One or more tester instruments may be shared among multiple devices; however, the FLASH 750 software manages the testing of the devices so these shared resources are used by one device at a time.

The FLASH 750 software manages and collects the test data from multiple parallel devices under test. Users can specify which sites are active. The tester and handler or prober are aware of the status of each site to ensure the data collected tracks the data from each device under test.

Components of a Test Program

Each component of a test program contains elements of test data and algorithmic programming. For example, even though most tests in a test program are based on data entered manually into a pre-defined spreadsheet, some custom action may be added. These added actions are defined in a programming language. This optional high-level code can be easily added by using the high-level interface.

The test data and algorithmic programming may be reused with or without modification; thus, all FLASH 750 test data and programming is accessible to the user for copying, editing, and cutting and pasting. In addition, the test data is available in an ASCII format, so it can be imported or exported to various tools.

Patterns

Each device is tested by a set of patterns or vectors.

The FLASH 750 software supports an ASCII pattern language which can include labels, microcode, comments, and the pattern data. Also, the parallel format is supported. The ASCII pattern files are compiled into a binary format for loading into the tester hardware.

Device Data and Limits

Each device has its set of operating data. Many test parameters are functions of the device data, which are usually expressed as high limit, low limit, and nominal values.

Device data is entered or imported into and debugged by a set of programming tools based on Microsoft **Excel**. The full capabilities of **Excel** are available in these tools; thus, the device data is interdependent and can be specified in a standard spreadsheet format.

Device Pin/Channel Mapping

Device pins are identified or mapped in three different ways:

- Logical pin name relative to the device
- Physical pin name relative to the handler/prober contact
- Tester channel

FLASH 750 users can refer to device pins in any of these three forms and can specify the form for data collection. Also, you can specify a logical pin without binding it to a tester channel, especially for devices with pads not bonded to their packages.

Device Pin-to-Tester Channel Mapping

A device pin map assigns the device pins to the tester channels and specifies other information, such as type of each pin: power, ground, input, output, and I/O.The device pin map is entered or imported into one of the programming tools based on Microsoft **Excel**. Device pins and groups of pins are available to the other spreadsheets as input parameters.

The device pin-to-tester channel mapping assigns a device-oriented name to the physical tester channels. The FLASH 750 device pin-to-tester channel mapping is not embedded in the test program. It is decoupled from the patterns because the same test program may test the same device in several different packages.

Multiple devices are tested in parallel by assigning more than one tester channel to each device pin in the pinmap. Each site is assigned a different tester channel for each pin.

Device Pin Groups

For convenience, users may group device pins together. For example, a 16-bit parallel data bus is much easier to treat as a single entity for patterns, timing, and levels than as individual pins. They are groups of device pins; they are not related to the test system instruments at this level. Users can then name specific groups of device pins. The same pin groups may be used with different device packages, which might be mapped to different physical tester channels.

Test Instances

A test instance specifies how a combination of pattern, timing, device data, and pinmap data in the test program is applied to a device. Test instances are defined in one of the programming tools based on Microsoft **Excel**.

Timing and Formats (Period and Edges)

Each device has timing data that specifies the relative placement of digital edges, period, and read strobes. Typically, multiple sets of timing values are assigned to a single device. Different sets of timing values are used for different tests. Timing data can be entered manually in the spreadsheets, or it can be entered, imported, viewed, and debugged in the timing/format editor.

In addition to the timing/format editor, you can use a waveform editor in **PatternTool** to view the timing data when it is combined with the pattern data. The expected and actual views of the data are displayed. You can navigate in this editor from the waveform view of the timing back to the source of the timing data.

Test Program Flow

A test program has a hierarchy or test flow, which defines which test instances should be executed and in what order. The test flow often includes binning instructions for each test. Pass and fail limits are typically derived from the device data. The FLASH 750 lets you specify this information by using the graphical program development tools.

FLASH 750 includes tools to graphically represent the test flow. Users can manipulate the program flow by using the *drag-and-drop* technique. This graphical representation is converted to code by the tools; thus, the coding is transparent to the user. However, you cannot insert arbitrary code at arbitrary points in the program flow. Using this method, code is inserted at defined interpose points only.

Operating System

System Controller

The operating system for the system controller is Microsoft **Windows NT**. All graphical display and interface tools, and all data processing software are executed by the system controller.

Per-Station Controller

Windows CE is the operating system for the per-station controller. The per-station controller operates each test station asynchronously, with little communications overhead between itself and the processor in the system controller.

The main tasks of the **Windows CE** OS:

- Manipulate the test data

  A local processor for each station means the programmer configures one device only. For up to 32 devices per system, the programmer never has to manage more than one device because the test program is copied to each local processor, and local processors typically do not communicate with each other.

- Execute the test flow defined in the test templates

  The test program resides on the local processor. It schedules the single tasks defined in the test flow that run to completion. Because the system has a per-station controller architecture, the programmer does not have to manage the complex test flow for multiple bin results, enable and disable multiple sites, or manage redundancy processing for multiple sites.

- Manage the station-assembly hardware

  FLASH 750 provides an API for low-level control of the station assembly for users to optimize the execution and flow control of the test program.

Programming Languages

System software and tools: C++

Custom production interfaces: Visual C++

FLASH 750 System Software

Overview

This section describes the following components of the FLASH 750 system software:

Patterns

The digital patterns from the pattern generator contain information that test the memory locations of a DUT. A pattern consists of microinstructions and optional initial conditions, which initialize the registers and the pattern generator mode at the start of a pattern. The address, data, and clock information is supplied by symbolic references to a timing set and a functional set defined in the **Excel** spreadsheets. The defined patterns are then compiled and linked with the test program.

Pattern Language

Pattern data is defined by a proprietary language that supports two file formats:

- Binary

  This format is used mainly by the test system; see Figure 2-1 on page 2-10. The pattern compiler creates a binary file when the pattern information is extracted from the templates,and is compiled,and linked with the test program. Pattern tools read and write patterns in the binary format. The instrument drivers also read the binary pattern data when loading data into the test system hardware.

- ASCII

  Users create ASCII pattern files and then compile and link them with the test program. Pattern tools can read from and write to pattern files in the ASCII format. The pattern information in the **Excel** spreadsheets is in an ASCII format, so users can exchange the pattern data.

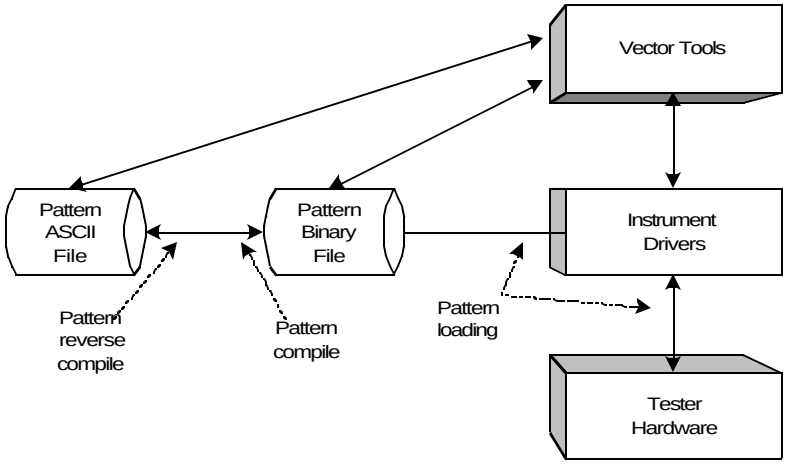

Figure 2-1. Pattern file formats

Instrument Drivers

Instrument drivers are coded by using C++ object-oriented techniques. The User Level API accesses the instrument drivers through their member functions and variables; see Figure 2-1 on page 2-10 and Figure 2-3 on page 2-15.

Resource Drivers

Resource drivers provide the functionality for testing devices, not for the tester hardware. Some of these utilities are accessible in the User Level API; see Figure 2-3 on page 2-15.

User Level API

The User Level API is a key component of the FLASH 750 system software: all test system hardware is accessed through it; see Figure 2-3 on page 2-15.

The FLASH 750 high- and low-level tester interfaces allow a user to add code that directly accesses the test system instruments. The added code accesses the tester hardware via the User Level API, which is a set of defined library of C++ functions, will not change even if the underlying hardware functionality or specifications are changed; consequently, all user code written in VBA or C++ do not have to be modified as the underlying hardware evolves.

Test Program Interface: IG-XL™

IG-XL™, the test program interface for the FLASH 750, is based on Microsoft **Excel** spreadsheets.

Key Features and Functions

- Based on familiar software tools: Microsoft **Office**
- Test programming and debugging oriented toward the device
- Graphical test program flow
- Interface to design and test data can be customized
- Production user interface can be customized
- Data tools with integrated spreadsheets
- Related test programs supported in single test program suite
- Off-line stations can run **IG-XL**™ on **Windows NT** or **Windows 95/98** machines.

IG-XL™ Databases

Data is entered into **IG-XL**™ databases by entering the appropriate information in pred-defined spreadsheets or by creating ASCII database files. FLASH 750 provides a database language for users to create database files. Data can be edited in ASCII or created by using the tools, as shown in Figure 2-2 on page 2-12.

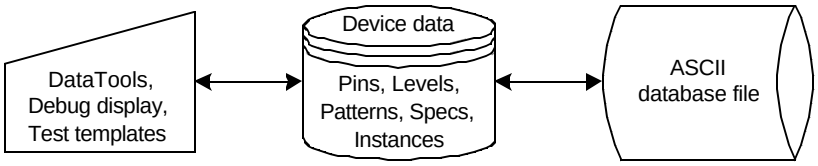

Figure 2-2. FLASH 750 IG-XL™ Databases

The **IG-XL**™ databases contain reusable device data; consequently, you do not have to recompile a test program after editing it.

IG-XL™ Component Groups

The **IG-XL** ™ software consists of the following groups of tools; see Figure 2-3 on page 2-15.

- Pattern tools:

  Pattern compiler

  Pattern editor

  Pattern debugger

- **Excel**-based tools:

  DataTool

  Debug display

- Production tools (Controls)
- Production interface tools
- Handler/prober communication drivers

IG-XL™ Component Forms

The **IG-XL** architecture is based on software component technologies. Software components are available in a several forms:

- Lowest-level component: C++ classes in source code.
- Intermediate level: **ActiveX** component in object form.
- Highest-level component: executable application. Applications, such as spreadsheets, word processors, databases, may provide a program interface through which the application itself is used as a component. The program interface standard used by FLASH 750 is OLE Automation.

  The first critical element is the test program process (item 1 in Figure 2-3). The entire test program is a single process consisting of the Microsoft **Excel** executable linked with the **ActiveX** object components and the C++ dynamic link libraries.

  Two methods communicate between the components within the test program process and between other processes and the test program: OLE (item 2 in Figure 2-3) and shared memory and **Windows** messaging (item 3, Figure 2-3).

DataTool uses Microsoft **Excel** as an application-level component (item 4, Figure 2-3). Because Microsoft **Excel** can be customized, the look and feel of the data on the spreadsheets is specific to the device test application. **DataTool** is the main interface for entering and debug device data and most other aspects of the test program.

Tester Instrument Debug Displays are **ActiveX** object components created in **Visual Basic** (item 5, Figure 2-3). The *Tester Instrument Debug Displays* show the hardware settings and allow a user to change the hardware settings. The debug objects are based on a single class interface that enables the *Exec* to manage them dynamically. This defined interface means that users can add displays to the system without modifying the existing software.

Pattern Tools use a C++ class component to implement the grid and manage the data management (item 4, Figure 2-3). These tools allow a user to edit and update the compiled pattern files and to view the failure information. They were implemented in C++ due to the amount of pattern data and the close interaction required with the *C++ Pattern Instrument Driver*. The volume of data shared between the *Pattern Tools* and *Pattern Instrument Drivers* determined that the *Shared Memory* and the *Windows Messaging* system was the most efficient communication medium between the two processes.

The **IG-XL** *Waveform Display* is another example of an **ActiveX** control written in **Visual Basic** (item 6, Figure 2-3). The same control is integrated into several different tools: (1) **DataTool** uses it for graphical display and editing of programmed timing data, and (2) *PatternTools* use it for graphical display of the nominal and actual waveform data captured on the test system.

The *Production Operator Interface* in **IG-XL** is unique: it must be easy for the user to customize. Thus, this interface consists of **ActiveX** controls that are available in the **Visual Basic** tool palette. These controls use an object-based interface (both GUI and program) to the test program (item 7, Figure 2-3).

The *Handler* and *Prober* communication drivers also use **Visual Basic**, but are **ActiveX** objects rather than **ActiveX** controls (item 8, Figure 2-3). The *Handler Control* dynamically loads and unloads these objects as the user selects different models of handler or prober.

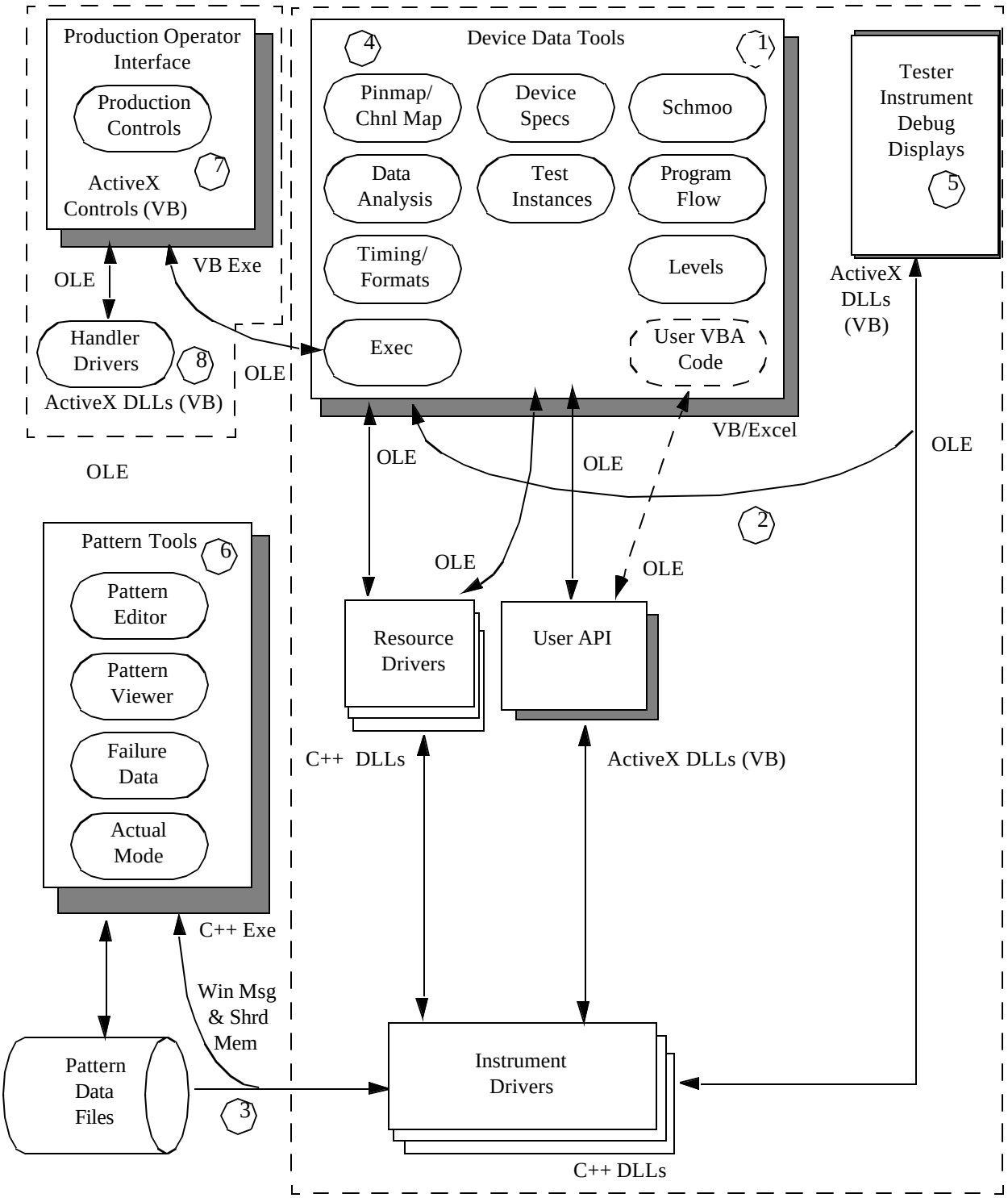

Figure 2-3. FLASH 750 IG-XL Architecture

User Interfaces

FLASH 750 provides several user interfaces for testing and debugging memory devices:

- High-level graphical user interface: pre-defined spreadsheets
- Low-level user interface: API
- Custom user interface
- Production interface

Test programs can be a mix the high-level spreadsheets and low-level API functions. The two are not mutually exclusive.

Most FLASH 750 user interfaces have a spreadsheet interface, like Microsoft **Excel**; thus, they can be integrated into an **Excel** workbook, template, or application.

Creating FLASH 750 Test Programs

FLASH 750 provides several ways for users to create a test program:

- FLASH 750 software tools

  To manually create a FLASH 750 test program, you must obtain a minimum amount of IC design data for the device to be tested. With this information, you enter the device information into the pre-defined spreadsheets. The tester software provides default values so you can quickly create a functioning test program. Once a test program is initially created, it can be revised to include all test requirements of the device.

- Copy and modify an existing FLASH 750 test program

  The usual way to develop a test program is to copy an existing program and modify it for the new device.

  Because of the significant architectural differences among Teradyne testers, such as Genesis, Pegasus, J9XX families, program conversion from these testers to the FLASH 750 is not feasible.

- Written manually

High-Level Graphical User Interface: Pre-Defined Test Templates

Purpose

The high-level user interface is a key feature of the FLASH 750; see Figure 2-4 on page 2-18. Users can develop and debug test programs by entering specific device data into pre-defined spreadsheets.

User API

The *User API* allows users to directly access the test system instruments by writing code containing statements in a defined language. This interface is independent of the hardware: it remains the same even if new hardware is added, the existing hardware is modified or the any system specification or functionality is changed. Consequently, the FLASH 750 *User API* allows a test program to directly access these test system resources and instruments.

Instrument Drivers

The software instrument and resource drivers, which are transparent in the high-level interface, are called from the test program rather than the standard test templates. The *User API* performs the actual device tests.

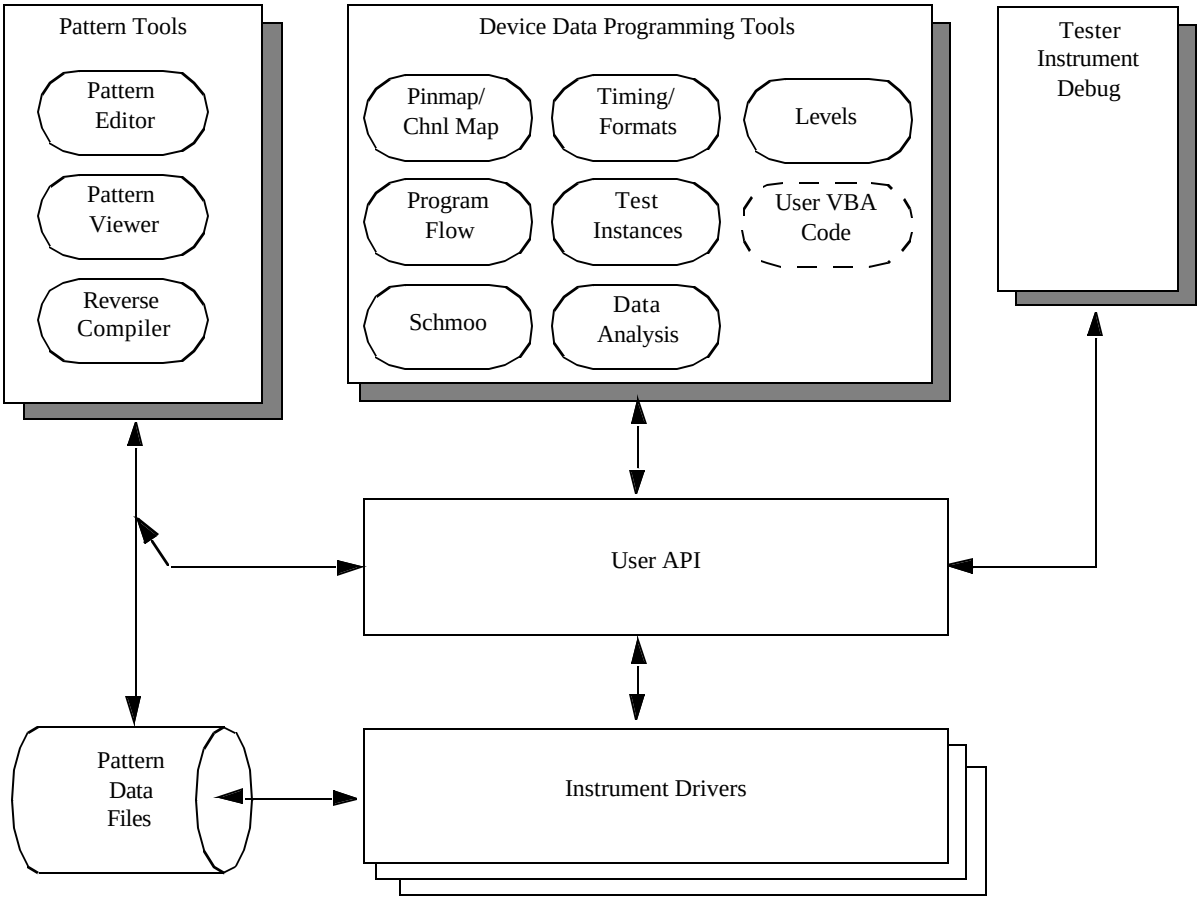

Figure 2-4. High-Level Graphical User Interface, Simplified Block Diagram

Low-Level User Interface: API

Purpose

Users can develop and debug test programs by directly accessing the *User API* with code written with defined API statements; see Figure 2-5 on page 2-19. This interface is similiar to the high-level interface except that custom test templates or user C++ code or both may have been added. You can use the low-level interface for the following purposes:

- Create custom test templates
- Modify existing test templates and save them as new templates
- Create arbitrary custom C++ code
- Create an executable test program

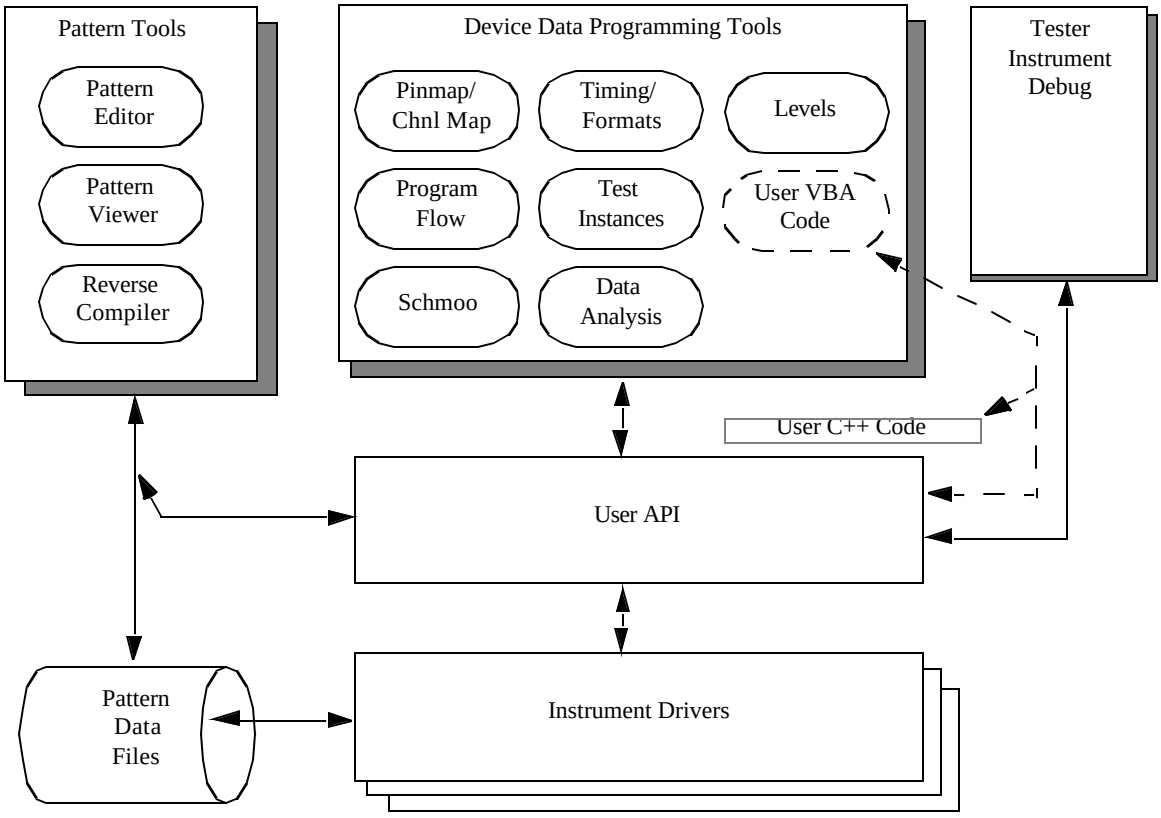

Figure 2-5. Low-Level User Interface, Simplified Block Diagram

Custom Test Program

Users with unique testing requirements can bypass most of the test programming environment and write an executable test program. The C++ code for this custom test program uses a utility library with a defined FLASH 750 API to access tester utilities and the *User API*; see Figure 2-6 on page 2-20.

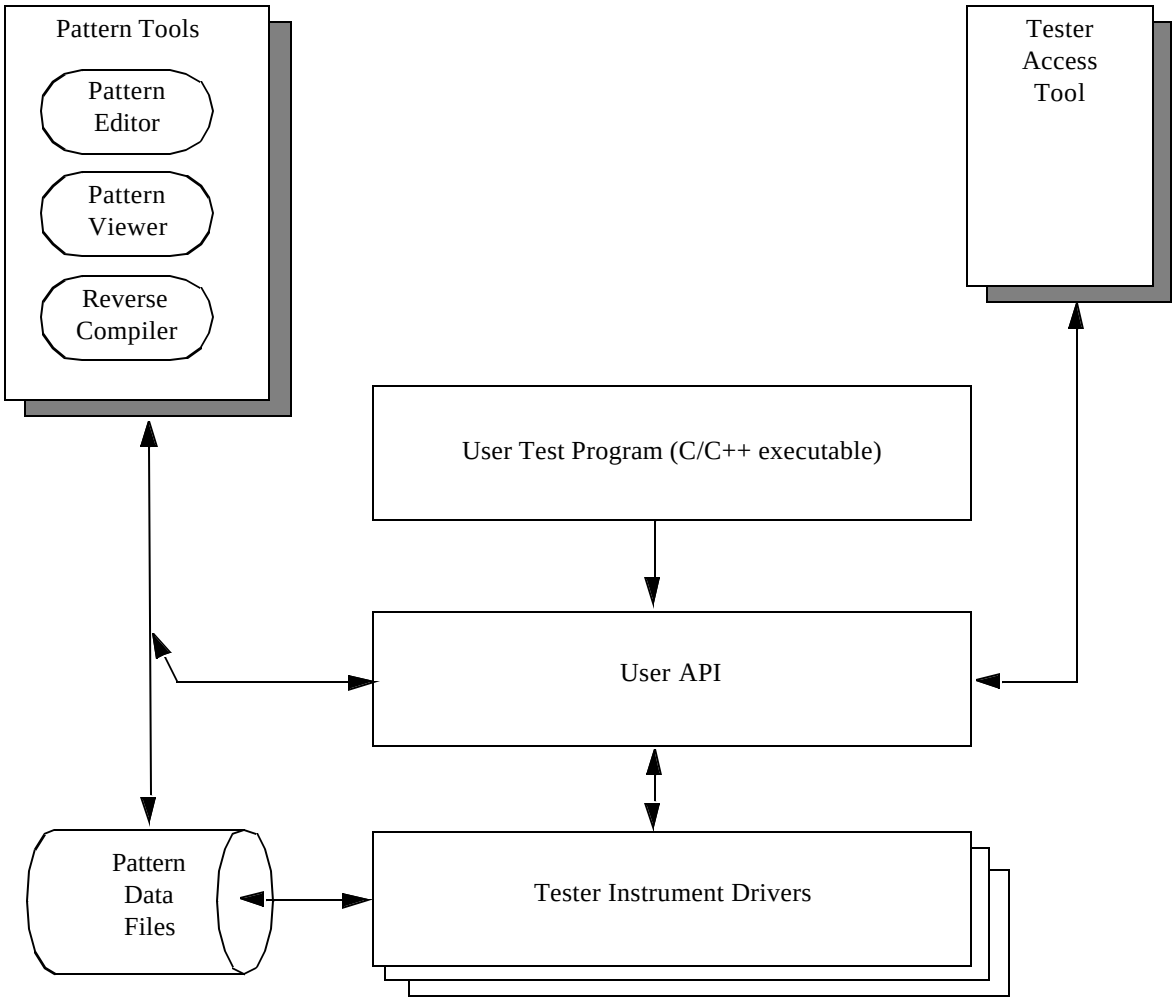

Figure 2-6. Custom Test Program, Simplified Block Diagram

Production Interface

Purpose

Memory device manufacturers require unique production interfaces for their ATE systems. A production interface allows an operator to control the tester while a test program is testing the production devices. FLASH 750 supports custom production user interface. All controls and status information for this interface are defined by a functional software library. The FLASH 750 simplifies the creation of a customized user interface by providing GUI test templates that users can customize with the **Visual Basic for Applications** (**VBA**) utilities and the FLASH 750 API.

Components

This interface has components similiar to the *Custom Interface* except that an **Excel** data engine and production controls are added; pattern tools and *Tester Instrument Debug* are removed. It is not intended for developing or debugging test programs because it has limited access to the tools for developing test programs; see Figure 2-7.

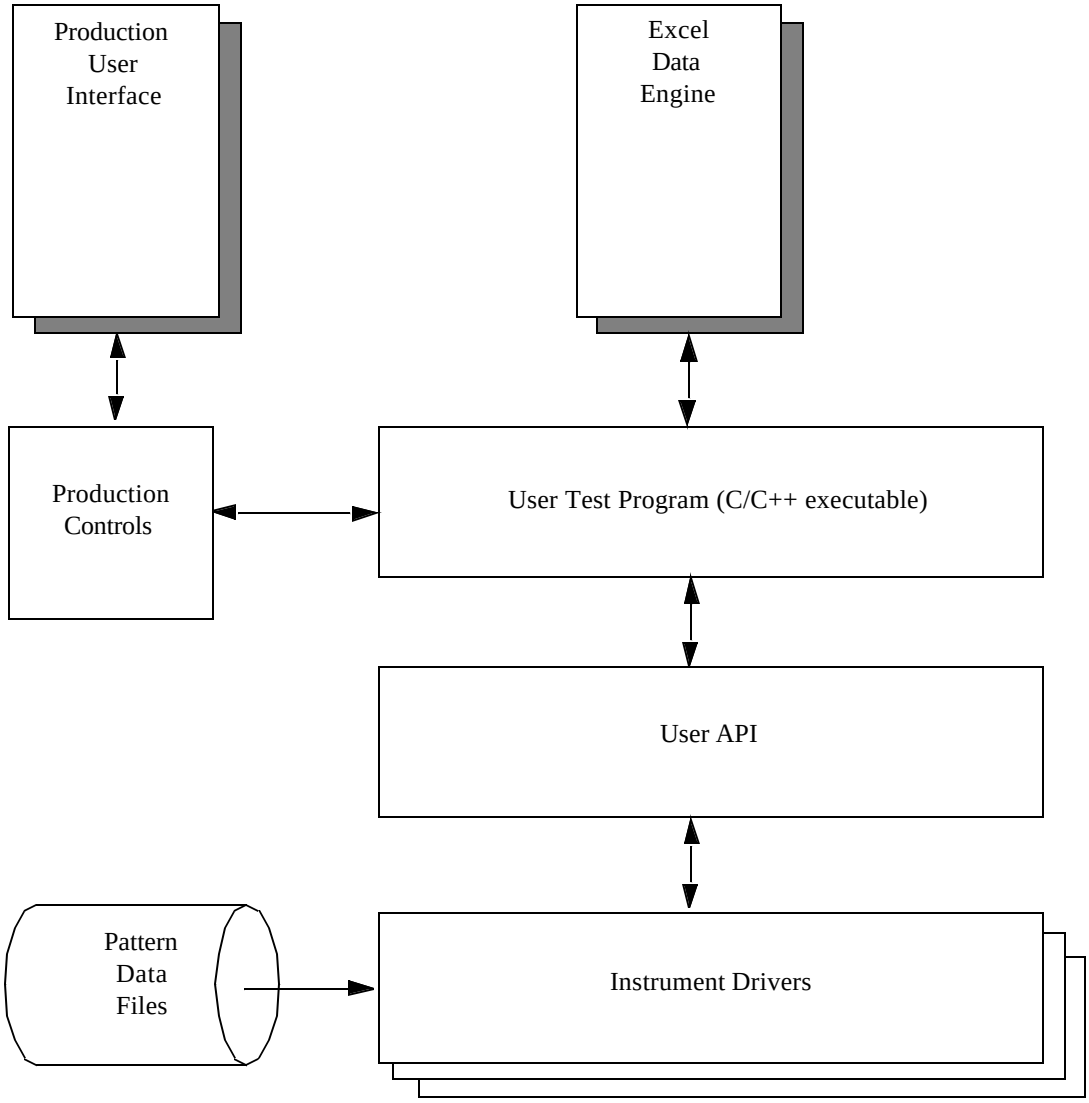

Figure 2-7. Production Interface, Simplified Block Diagram

Graphical Tools

Datalogging

Setting Up Data Collection

Data collection setup is associated with a specific test program; thus, data collection does not have to be set up each time the test program is run. Users can define an ASCII setup file that loads the data collection setup with a test program. In addition, you can modify this setup by using the debug tools, *User Level API*, or the test program.

Collecting the Data

Data collection examines the programmed data collection mode at the end of each test. The data is extracted from the software data structures or tester hardware and then stored. Several formats are provided for storing the data. Stored data is optimized for performance. While collecting the detailed data during device testing, FLASH 750 minimizes the decrease in throughput by caching the data in CPU memory for periodic writes to disk.

The amount of data collected varies during program development, debugging, and characterization; however, this large amount data must be moved to a database archive. During production testing, pass/fail data are typically collected; however, detailed data is often sampled during production to implement process monitoring and statistical process control (SPC).

Key Functions and Features

- GUI to manage the data collected from the user interface software and the test program.

  This **Excel**-based GUI is for data reduction and analysis during characterization and device debug. It is used to examine a selected portion of the data collected during debugging or characterization because **Excel** is not useful for examining very large amounts of data.

- Datalogging in ASCII, STDF, or **Excel** file formats

  Standard Test Definition Format (STDF™) is a binary data format read by Teradyne's FIRMS™ database.

- Full functional datalogging with flexible datalogging setup
- Search/shmoo in one-, two, and three-dimensions
- Histogram

A common method of analyzing test data is to plot a histogram of the distribution of the test results of one or more tests over a sample of devices. FLASH 750 provides a GUI tool, **DataTool**, to set up the collection of the appropriate data for generating histograms. Applications supplied with the tester, such a Microsoft **Excel**, allow you to generate histogram plots from **DataTool**.

- Summary report

  A summary report can be generated, which includes binning statistics of the completed lot, can be generated. The summary report can be saved to disk or sent to a printer.

- Log all failures

  FLASH 750 provides datalogging utilities to log the following test results:

  1. Testblock name or number or both
  2. Total repairable count
  3. Redundancy result for each device—repairable, no errors, unrepairable, redundancy analysis timed out, or unknown
  4. Redundancy segment:
  5. Redundancy segment number
  6. Number of rows replaced in the segment
  7. List of the rows replaced
  8. Number of columns replaced in the segment
  9. List of the columns replaced

Data Manager

Purpose

The *Data Manager*, a C++ library of classes, is the intermediate layer between the high-level GUIs and the *Instrument Drivers*. High-level GUI tools use the *Data Manager* to communicate with the drivers that load and manage the hardware. The *Data Manager* has no GUI; its operation is transparent to the user.

Key Features and Functions

- Transfers all data to and from the GUIs.
- Replaces, retrieves, stores, and validates the data in the **DataTool** sheets, including the load redundancy table, SDA table, error map, and scramble RAM values.
- Provides the debug tools with the required pin and channel map information.
- Examines the configuration files containing the information identifying the boards, their slot assignment, and calibration constants, such as the round trip delays and temperatures.
- Maintains a record of changes to redundancy table, SDA table, error map, and scramble RAM.

Debug Displays

Purpose

The debug displays help a test engineer to determine if a debugging problem is in the test program or in the device. By interpreting the device failure data of the FLASH 750 debugging display, a test engineeer without having detailed knowledge of the device can understand the implications of any particular failure.

Key Features and Functions

Users can manipulate the DUT data or the test system by using the appropriate features:

- Test-oriented trapping (trap on fail, trap on test)
- Full symbolic debugger
- Graphical setup for debugging tools
- Pattern debugging tools: edit vectors, reburst, loop
- Setup and display of pattern history capture
- Nominal and actual waveform displays
- Display and edit the tester vector memory

Displayed Data

Debug displays show the settings for all boards and modules:

- CalCUB (Calibration and Utility Board)
- Channel Board
- Memory Test Module
- Device Power Supply
- ECR/DBM

FLASH 750 failure data is available in ASCII format so the data can be exchanged among test engineers or the IC designer for analysis, or imported by software analytical tools:

- Channel numbers
- Pin names
- Physical names
- Repeat or Loop Counter
- Opcode
- Flags
- Timing Sets (number)
- Timing Sets (name)

Debugger for Control Flow Section of a Test Program

The control flow section of a test program is probably the most customized part of a test program. It is the part the test engineer understands the most; however, it is most likely to have require debugging. The FLASH 750 includes the following debugging features:

- Navigate through the code
- Stop on the failed test
- Stop on error
- Ignore failure
- Breakpoints
- Step into
- Step over
- Continue
- View the variables
- Execute the code fragments

FLASH 750 includes specialized features to debug control flow problems for devices tested in parallel. For example, you may want to ignore failures for only one particular site.

Error Capture

Purpose

Error capture helps users to develop and debug redundancy analysis algorithms and test programs and to acquire operating data about the DUT.

Key Features and Functions

- Create, view, and edit the device failure data
- Create, view, and edit the contents of the ECR and DBM
- Load and interactively execute the task from the buffer error memory
- Execute redundancy analysis and view the results
- Execute SDA analysis and view the results

BitMapView—Bitmapping

FLASH 750 provides a graphical bitmapping tool, **BitMapView**, for users to view the failure data for a given set of conditions in device geometry. It displays an accurate representation of the device topology by showing a one-to-one correspondence between the displayed image and the DUT. Consequently, this tool helps users in identifying any design and process failures of the device. Bitmap plots can be initiated from within the test program; thus, each die or test site can be plotted during test execution to speed up and automate device characterization.

Because the size of the displayed data may be very large, FLASH 750 includes the following features for navigating through and viewing all failure data:

- Edit the bitmap display parameters
- Select the area to display
- Zoom in or out of an area
- View the location of the display area in relation to the area of the entire device
- Show redundancy analysis results: rows and columns targeted for replacement.
- Status information
- Reset the redundancy results in the display
- Simulate redundancy processing
- View the bitmap data previously stored

bemsee

bemsee presents users a view of the data in the arrays of the DBM or ECR. Compared with bitmapping, **bemsee** shows the data according to its physical locations in the tester hardware, not the device topology.

Features include:

- Select the station to display
- Select the RAM to display: main RAM array, scramble RAM, or error RAM
- Display the contents of the ECR RAM
- Display the lowest X and Y address of cells in the current display
- Navigate forward and backward along the cells in a row or column one cell at a time or one block at a time
- Edit the data in the RAMs
- Search for the first or next error location by row or column in the RAM
- Set the lowest X and Y address of cells to be displayed
- Update the display with data from the tester hardware

Bitmapping on Wafer

BitMapView includes a graphical tool, bitmap on wafermap, for users to view failures from all die on an entire wafer.

Parametric Bitmap

The parametric bitmap feature saves the test results to the ECR.

Redundancy Analysis

The FLASH 750 provides on-the-fly and processing of the redundancy analysis results. Up to 256 segments of a device, 32 rows and 32 columns, are supported.

FLASH 750 software includes several different algorithms for redundancy analysis. Comprehensive algorithms examine many or all potential solutions to determine which are viable:

- All—examines all potential solutions. For relatively simple redundancy schemes or low-density chips, All algorithms can complete the analysis in acceptable times
- Best—finds the viable solutions and then scores each one to select the best algorithm. Best methods are precise and thorough, but, for regions with many spares, can be relatively slow.
- Most—a fast, once-through approximation that generates a solution in a single pass. It has an acceptable level of accuracy; however, this algorithm may fail sometimes to find a solution, even though one exists.

SchmooTool

ShmooTool is a graphical tool for debugging a test program and characterizing a device. It produces two- and three-dimensional plots, called schmoos, of the device characteristics. A shmoo is a graphical view of the results of margining one (one-dimensional shmoo) or two (two-dimensional shmoo) test parameters. Shmoo plots may include other data for each point, such as the pattern fail address. A schmoo plot helps users to visualize the pass/fail relationships between test parameters. By determining the passing limits of voltage, current, timing, or other device parameters from a schmoo, you can optimize a test program for a specific application.

WaferMapView—Wafermapping

Definition

WaferMapView captures information about the test results by die coordinates of a wafer and then prints the display data in an X-Y coordinate format, which is a view of the wafer. Each die is represented on the wafermap, including the binning information. The data is displayed in one of the five different modes:

- Graphic displays, such as bitmaps
- Numeric displays with up to 4 fields
- Coded displays
- String displays
- Colored textured displays

Purpose

FLASH 750 provides two methods to test whether a die is good or bad:

- inking—the prober marks the bad die with a spot of ink.
- inkless binning—FLASH 750 remembers which die on the wafer are good and bad. It saves this information so only good die are packaged later in the process.

In both cases, testing at wafer probe focuses on the wafer as a whole and on the die on each wafer. The operator needs to view the wafer status, which die have been tested, and the binning of each tested die. This information can be determined by viewing a wafermap of the device.

Key Features

A FLASH 750 wafermap is updated during testing; thus, it is also known as real-time wafermapping.

Wafermap files may be saved or printed in an ASCII format of a map or a list.

DataTool

DataTool is the main GUI for creating a high-level test program. It is a set of pre-defined Microsoft **Excel** spreadsheets for users to enter the device parameters and to allocate the tester resources for a desired test. The information in these spreadsheets is combined by the FLASH 750 software to create an executable test program; see Figure 2-8.

Spreadsheets

Purpose

The test program data is contained and displayed in a set of **Excel** spreadsheets. Device data, except for pattern data, is extracted from the **Excel**-based spreadsheets and stored in data structures accessible to the instrument drivers. These data structures are created when the test program is loaded, and if the device data is updated during device testing.

Workbook

A workbook is a collection of spreadsheets that display the programming information for a single program or a set of related programs. All information for executing a program is in the workbook except for the test patterns, which are managed by the pattern tools. A workbook may contain data used by more than one test program. Although test programs may share spreadsheets; a test program, however, cannot consist of spreadsheets in more than one workbook.

The relationship between the test templates and a workbook is shown in Figure 2-8 on page 2-34.

.

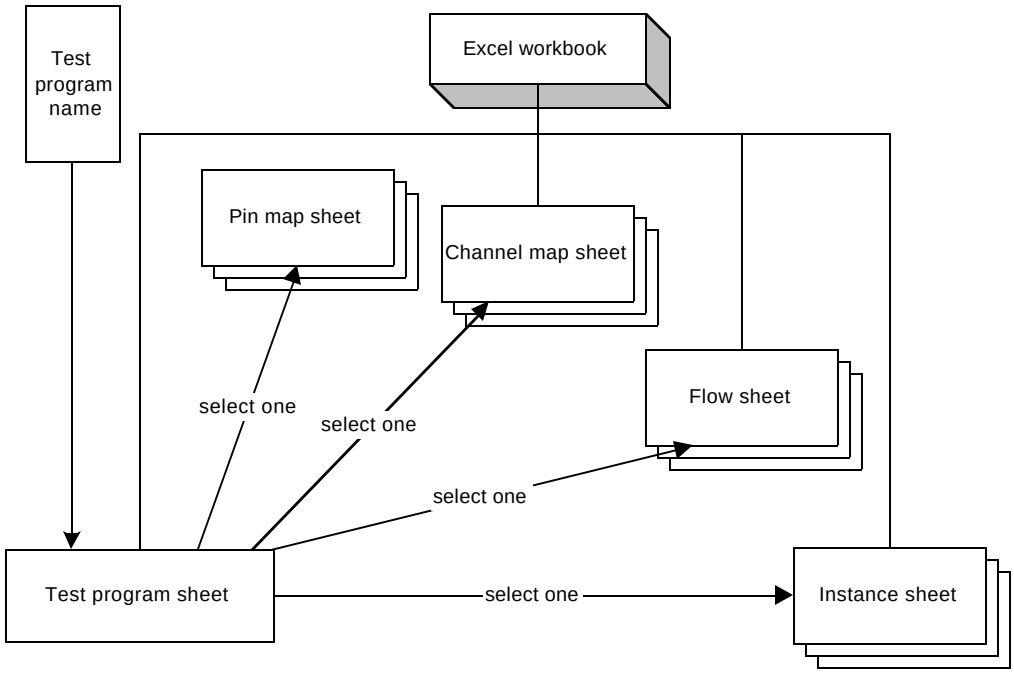

Figure 2-8. FLASH 750 Test Templates, Simplified Block Diagram

Categories of Spreadsheets

Spreadsheets are grouped into two categories: test program global sheets and Instance local sheets.

Test Program Global Sheets

A test program has one set of global sheets that are shared by all instances (tests) in a test program. A workbook may have more than one version of each type of global sheet, but only one of each type may be selected for a test program.

Users define a test program sheet by selecting the following global sheets:

1. Pinmap sheet—required—assigns the logical names and channel types to the device pins and assigns the device pins to the pin groups.
2. Channel map sheet—required—maps the logical device pins to the physical tester channels. For multisite test programs, the device pins are mapped by site to the tester channels.
3. Instance sheet—required—names the templates and instances of those templates in a test program. The names of instances are the tests executed in a test program. Other local instance sheets that specify the timing, levels, and patterns are selected for each template; thus, an instance sheet is the link between the test program global sheets and the instance local sheets.
4. Flow sheet—required—determines the instances executed in a test program, the execution order, and the corresponding test number. It can also specify the execution logic and binning.
5. Pattern group sheet—optional—specifies the patterns in the named pattern groups. Patterns in a pattern group are loaded contiguously into the tester memories. This sheet is optional unless it is mentioned in a pattern set sheet.
6. Test program sheet—optional—selects the other global sheets in the test program by naming them; one name per type of global sheet in one row per test program. If one copy of each type of the required global sheet is present, the test program sheet itself is not required.
7. Redundancy table sheet—optional—defines the parameters for initializing the redundancy table. The redundancy table is a software structure of the device topology parameters. The redundancy software uses this table to analyze the DUT. Only one redundancy table sheet is allowed because most test programs are written to test one type of device.

8. SDA (Statistical Data Analysis) table sheet—optional—defines the parameters for initializing the SDA table. This table is a software structure containing the device topology parameters and row/column/segment error thresholds. The SDA software uses this table to analyze the DUT.
9. Error sheet—specifies which ECR data channel receives the failing data from an I/O pin. This sheet is required when the a test specifies error capture. More than one sheet is allowed so a different error map can be defined for different test instances. Defined parameters include the data compression mode or the data compression ratio (which can determine the compression data width), and the ECR data channels receiving the failing data from the I/O channels.
10. Scramble RAM sheets—optional—define the address scrambling during error capture and redundancy analysis.

Instance Local Sheets

Each instance or test is assigned to the instance local sheets. These sheets are shared by instances, but can be unique to each instance. The names of all instance sheets selected by an instance comprise all or some of its arguments and determine the values of edge timing, period, and levels for the channels and the patterns in a test. A typical test program has more than one version of each type of local sheet. A workbook may have many types of local sheets, with only a subset selected by the instances in a test program:

1. Edge set sheet—required—defines the named sets of the edge timings and formats for the channels.
2. Timing set sheet—required—defines the named timing sets by assigning the edgesets and periods to pins and pin groups.
3. Level sheet—required—assigns the values to the key drive and compare parameters for the device pins and the tester resources.
4. Pattern set sheet—required—specifies the pattern groups and patterns used by an instance, starting order, starting locations in patterns, and which patterns supply subroutines compared with those started explicitly.
5. Spec sheet—optional—assigns values to the symbolic variables used by other sheets. The symbolic information has no representation outside of FLASH 750.
6. ECR/DBM sheet—required for capturing errors with the Error Catch RAM (ECR) or for using the data buffer memory (DBM).

PatternTool

Purpose

PatternTool provides several GUI tools for compiling, reverse compiling, managing, editing, and displaying patterns:

1. Pattern editor—is a Microsoft MFC-based utility that reads an ASCII pattern file for editing. The data includes information about the pattern vectors, pin maps, pin names, and other parameters.
2. Pattern compiler—compiles the ASCII pattern files into FLASH 750 binary pattern files. It is invoked from a command prompt or executed as a Windows GUI program; see Figure 2-9.
3. Pattern reverse compiler—creates an ASCII pattern file from a FLASH 750 binary pattern file. Unlike the pattern compiler and the pattern compiler, the pattern reverse-compiler is invoked from the command line only.

Key Functions and Features

- Read data from the pattern data file or from the test system hardware
- Combine the pattern data with other data, such as timing, for displaying the digital waveforms and failure data
- Display and modify the digital patterns including microcode, labels, and comments
- View the parallel or scan data or both
- Read from and write to the data files on disk
- Read from and write to the tester memory
- View the data in the programmed order and overlay fails
- View the data in the execution order, overlay fails or state (low, high, mid) (History RAM), or overlay fails and state
- View all data or fails only
- Navigate between the vector modules
- Insert and delete the vectors
- Patch the vectors
- Mask the failures by pattern or by channel
- Support the *learn mode*
- Trigger on the vector and fail
- Loop over the pattern or module
- Capture all failures by automatically rebursting pattern
- Display how tester memory is used (both LVM and SVM)
- Support the oscilloscope sync signals
- Support off-line simulation of patterns

Figure 2-9 on page 2-38 shows the relationship between the pattern tools and the rest of the system.

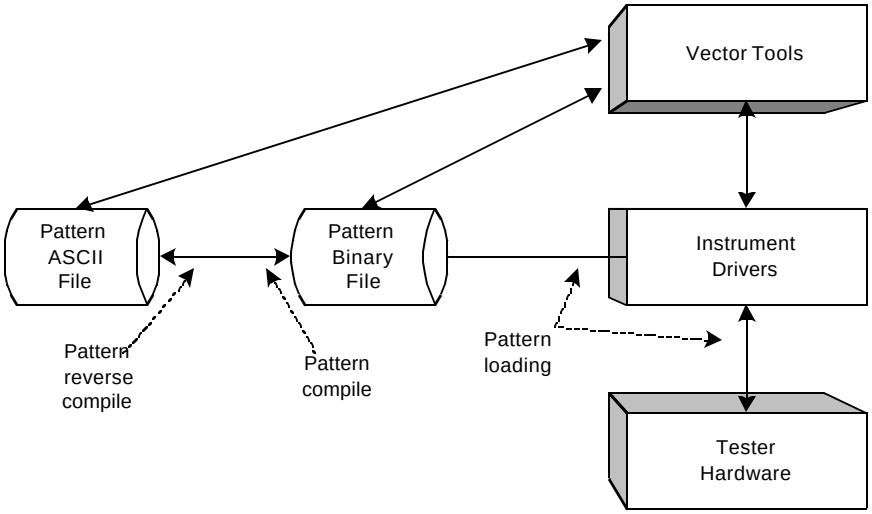

Figure 2-9. Pattern Tools in FLASH 750 Architecture

Tester Instrument Debug

While the pattern tools and test program development tools are device-oriented, high-level user interfaces, the *Tester Instrument Debug* is a low-level interface for users to view and modify the test system hardware directly. Without knowledge of the rest of the test program and without knowing the assigned pin name to the channel or the labels associated with the pattern data, you can access the test system hardware directly to examine the pattern data for a particular channel or to modify a programmed value of a tester instrument. By directly accessing the test system hardware, users can verify what is actually happening in the tester and edit the data during debugging.

Even though the *Tester Instrument Debug* allow you to manipulate the tester hardware, it accesses the tester hardware through the *User API*. Even though this tool directly accesses the tester hardware, the specific hardware is not exposed. Each instrument is presented as an abstract virtual instrument through the generic (hardware independent) tester interface.

Timing/Format Editor

Purpose

A timing/format editor allows users to view and modify the timing sets and assign the timing sets and their formats to the device pins. The timing information is formatted like an **Excel** spreadsheet.

Key Functions and Features

- Read from and write to data files on disk
- Read from and write to tester memory
- Display the data in graphical form (logic display/waveform display)
- Support the actual mode in coarse and fine resolutions
- Navigate from displayed waveform to timing data for programmed waveform.
- Modify the timing for a particular cycle (isolate cycle)
- Support the propagation delay measurement (edge find)

Parallel Test

Parallel test feature support *serial blocks* in C++ and true parallel match looping.

Test Looping

Looping a test has two purposes:

- Help analyze the intermittent failures by looping a single test
- Obtain a synchronization signal for an external scope to measure the DUT signals

Features and Functions

- Summary tables of menu selections that have been run, and the pass/fail percentage of each test category.
- Datalogging modes include: datalogging all tests, only failing tests, only tests with results that exceed specified limits, or a list of tests performed.
- *Period* mode for specifying the operating period, thereby checking the test system under a slower operating speed.
- *Learn* mode allows a sequence of commands to be stored in a default command file, which can be executed at a later time.

System Diagnostics

System diagnostics are a set of programs that verify the functions of the test system across all possible programmed values.

System diagnostics may take several hours to verify the entire system; however, they can be configured to diagnose only a specific fault detected by **Checkers**. Failures identified by system diagnostics indicate the source of the failure. The execution and results of system diagnostics are automatically logged.

Maintenance Log

This log is accessed through a system maintenance user interface, which guides the maintenance technician through the maintenance process.

Miscellaneous

Off-Line Test Program Development (Simulation)

The ability to develop and debug a test program without actually using a FLASH 750 is an important feature because a test system in a production environment is not always available for developing and debugging a test program. The FLASH 750 system software system can run on any **Windows 95/98/NT** computer that is not part of a test system. Note that execution of the test system software may require **Windows NT**.

Multiple Test Stations

The FLASH 750 is inherently a single test station; multiple test stations are not currently supported.

Performance is a critical to a memory test system. The following performance benchmarks were major factors in designing the FLASH 750 hardware and software architectures:

- Test execution time
- Loading time
- Interactive performance
- Calibration time
- Tester utilization