Host-Target Testing

Executive Summary

This paper discusses the issues involved in host-target testing, and shows how AdaTEST and Cantata can be used to implement an effective host-target testing strategy.

The efficiency of host-target software development can be maximised by an effective host-target testing strategy. Target system bottlenecks can be avoided and investment in expensive target resources such as in-circuit emulators can be minimised.

IPL is an independent software house founded in 1979 and based in Bath. The company provides a comprehensive range of software services and also supplies the AdaTEST and Cantata software testing and verification packages. IPL was accredited to ISO9001 in 1988, and gained TickIT accreditation in 1991. AdaTEST and Cantata have been produced to these standards.

Copyright

This document is the copyright of IPL Information Processing Ltd. It may not be copied or distributed in any form, in whole or in part, without the prior written consent of IPL.

IPL
Eveleigh House
Grove Street
Bath
BA1 5LR
UK
Phone: +44 (0) 1225 475000
Fax: +44 (0) 1225 444400
Email: tools@ipl.com

Last Update:08/06/2011 09:23:00

1. Introduction

Since the advent of high level languages, the practice of developing software in a different environment to the environment in which it will eventually be used has become common. The development environment is referred to as the **host**, and the environment in which the software will be used is referred to as the **target**. Such a development strategy is referred to as **host-target development**, and the associated testing practices as **host-target testing** or **cross-testing**.

Traditionally, host-target development has been used for embedded systems, where a powerful multi-user host environment is used to develop software which ultimately executes in an embedded microprocessor target environment. The PC explosion has opened whole new avenues for host-target development, with PCs being used as a development host for embedded systems, and also as a host to develop software which will eventually execute on mini or mainframe systems.

This paper discusses the issues involved in cross-testing, and shows how AdaTEST and Cantata can be used to implement an effective cross-testing strategy.

Section 2 of this paper explains why it is necessary to test in the host environment. Testing strategies for host-target development are discussed in section 3, with attention to unit testing, software integration, and system testing. Section 4 looks at the implementation of host-target testing. Section 5 shows how AdaTEST and Cantata can be used to implement a cross-testing strategy, followed by conclusions in section 6. Annexe A discusses host-target communication options.

2. Why Test in the Host Environment?

When developing software using a host-target environment, the question of which software should be tested in the host environment and which software should be tested in the target environment becomes important. Theoretically, all testing should be conducted in the target environment. After all, it is the target environment in which the software will eventually be used. However, constraining all testing to the target environment can result in a number of problems:

- A bottleneck may form of many developers competing for time on the target environment in order to test software. To alleviate this bottleneck, developers will require more target environments.
- The target environment may not yet be available.
- Target environments are usually less sophisticated and less convenient to work with than host environments.
- Target environments and associated development tools are usually much more expensive to provide to software developers than host environments.
- Development work may interfere with the continued use of existing applications in a target environment.

Economics and development efficiency therefore provide good reasons to conduct as large a proportion of the software lifecycle in the host system as possible, including software testing.

©IPL Information Processing Ltd

3. Testing Strategies

When planning software testing in a host-target environment, a software developer should begin by addressing the following questions about the development:

- How many developers will be involved in each of unit testing, software integration, and system testing?
- How much software is there to test, and how long will it take to test it?
- What software tools are available in the host environment and in the target environment, how expensive are they, and how suitable are they?
- How many target systems are available to the developers, and when will they be available?
- What means of connecting between host and target are available?
- How quickly can software be downloaded to the target environment for testing?
- Are there any constraints on the use of host and target environments placed by standards (such as safety critical software standards)?

It is rare for a host environment to be identical to a target environment. There may just be minor differences in configuration, or there may be major differences such as between a workstation and an embedded control processor, where the two environments may even have a different instruction set.

With answers to these questions, and using the guidance of this paper, a developer should be able to plan a host-target testing strategy. The general objective is to conduct as much testing as possible in the host environment, but only while this saves time in the target environment.

Unit Testing

All units which are not specific to the target environment can be unit tested in the host environment. Only a few target specific units will require unit tests to be run directly in the target environment. The proportion of software tested in the host environment can be maximised by making the target specific units as small as possible and localising access to all target specific interfaces.

Running tests in the host environment will be much quicker than running tests in the target environment while the tests are being developed. When a unit test has been completed in the host environment, a simple confirmation test may then be conducted by repeating the test in the target environment. This will verify that test results have not been influenced by differences between the host and target environments.

Confirmation tests in the target environment will identify unknown, unanticipated or unaccounted differences between the host and target environments. For example, there may be bugs in the target compiler which are not present in the host compiler.

©IPL Information Processing Ltd

Software Integration

Software integration may also be achieved largely in the host environment, with units specific to the target environment continuing to be simulated in the host. Repeating tests in the target environment for confirmation will again be necessary. Confirmation tests at this level will identify environment specific problems, such as errors in memory allocation - deallocation.

The practicality of conducting software integration in the host environment will depend on how much target specific functionality there is. For some embedded systems the coupling with the target environment will be very strong, making it impractical to conduct software integration in the host environment.

Large software developments will divide software integration into a number of levels. The lower levels software integration could be based predominantly in the host environment, with later levels of software integration becoming more dependent on the target environment.

System Testing and Acceptance Testing

All system tests and acceptance tests have to be executed in the target environment. This may be facilitated by developing and executing system tests in the host environment, then porting and repeating them in the target environment.

Target dependencies may prevent system tests developed in a host environment from being repeated without modification in a target environment. Careful structuring of software and tests will minimise the effort involved in porting system tests between the host and target environments. However, fewer developers will be involved in system testing, so it may be more convenient to forgo execution of system tests in the host environment.

The final stage of system testing, acceptance testing, has to be conducted in the target environment. Acceptance of a system must be based on tests of the actual system, and not a simulation in the host environment.

4. Test Implementation

Having looked at overall testing strategies involving cross-testing, we will now look at specific issues relating to the implementation of cross-testing.

Software Architecture

Cross-testing requires software to be as portable as possible between the host and target environments. In order to achieve this objective, differences between the host and target environment must be identified, understood, and either avoided or isolated by the software designer. It is the level of success at achieving this portability which governs the extent to which cross-testing can be used.

There are a number of areas where software may interface to the target environment:

- Direct access to target hardware, such as input-output devices.
- Calls to target environment software, such as an operating system or real time kernel.

©IPL Information Processing Ltd

Interfaces to the target environment should be isolated in separate units, removing target environment dependencies from all remaining software, as illustrated in figure 4.1.

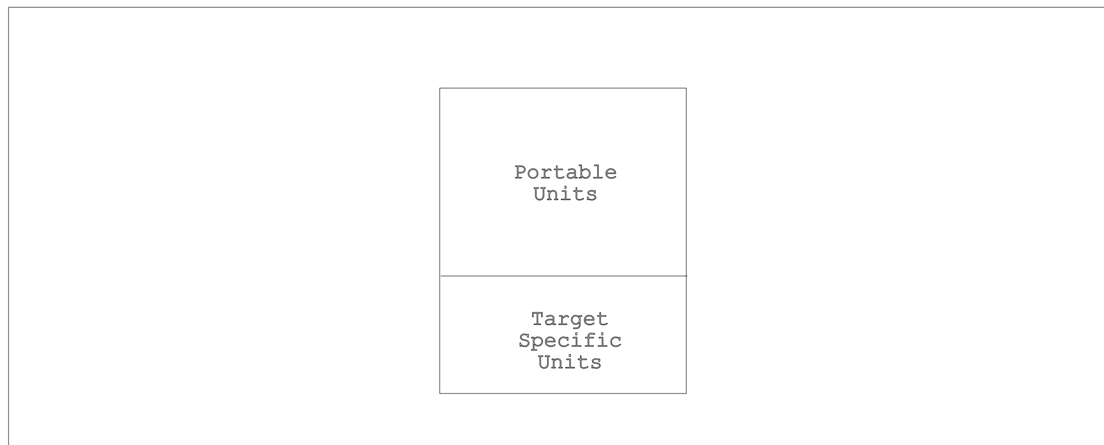

Figure 4.1 - Design Structure for Cross-Testing

There are also a number of hidden potential dangers, which may arise from differences in the fundamental attributes of the host and target environments:

- The ordering of bytes within words.
- Word length.
- Structuring and packing of compound data such as arrays and records.
- Data representation

Once a designer is aware of these differences, it is relatively simple to design software which is immune to them.

Finally, there may be differences in library routines which are supposed to be standardised. For example, C input-output library routines such as printf and scanf often vary between compilers. Problems can be avoided by isolating the use of such routines to target specific units. Alternatively, software can be designed to be immune to differences in library routine behaviour.

These design guidelines are not unique to cross-testing. Isolation and avoidance of environment dependencies are generally considered to be good design practice, so just designing with cross-testing in mind can improve the quality of software.

Simple Test Structure

The simplest test structure assembles the software under test, the test drivers and the test stubs into a single executable test program. The test driver calls the software under test, the software under test executes (possibly calling stubs) and then returns control to the test driver. This process is repeated for each test case.

This simple test structure is suitable for all unit tests, and for integration of units up to a point where the software architecture introduces concurrency (such as multiple tasks).

©IPL Information Processing Ltd

The degree to which stubs are used will depend on whether a `bottom-up' or `isolation' unit testing strategy is being followed. However, to achieve portability between the host and target environments for cross-testing, all target specific units will have to be simulated by stubs.

In any software development where concurrency is not involved, the simple test structure can be used right up to system testing.

Concurrent Test Structure

Concurrency is playing an increasing role in the architecture of software. The concept of multiple tasks was originally the preserve of real time systems. The growth of software communications, networks, client-server systems, and general increase in the complexity of problems which software is used to solve, has resulted in an increase in the use of concurrency in software designs.

For example, a few years ago there was little call for the PC operating system (DOS) to support multiple concurrent tasks. Requirements have now changed, with the successors to DOS (Windows, OS2, UNIX) all supporting multiple tasks.

For Ada developments, concurrency can be contained within an Ada program, so the simple test structure discussed above can be used. When developing software in C and C++, the simple test structure can be only be used when testing the units within a task, and for levels of integration up to task level.

To test C and C++ software at task level, and at levels involving multiple tasks, a concurrent test structure is necessary. The concurrent test structure can also be used to test Ada software involving multiple Ada programs.

Consider a C task designed to operate in the target environment, which has been structured according to the guidelines given in this paper (see figure 4.2).

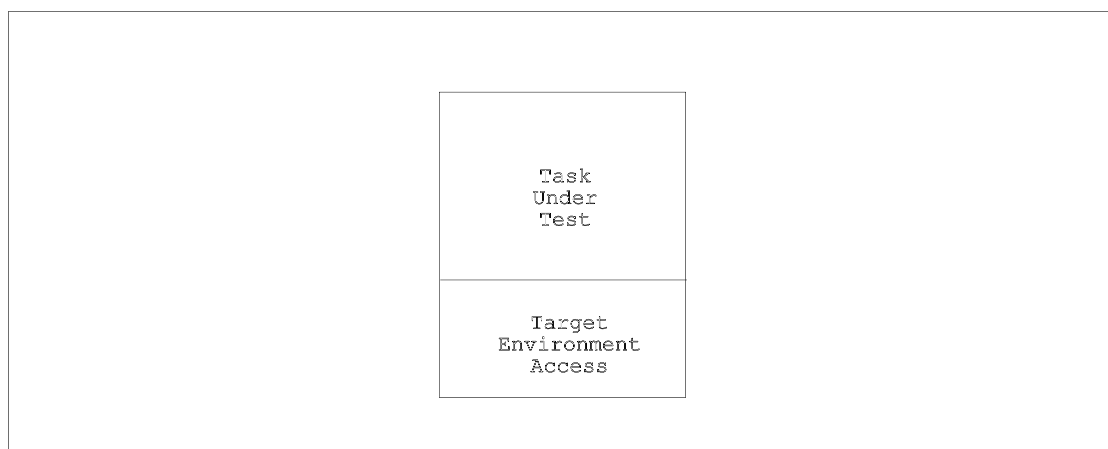

Figure 4.2 - Task Structure

A concurrent test structure for this task, in the target environment, would build the test driver into a another task, communicating with the task under test through a specially built target test environment access, as shown in figure 4.3.

©IPL Information Processing Ltd

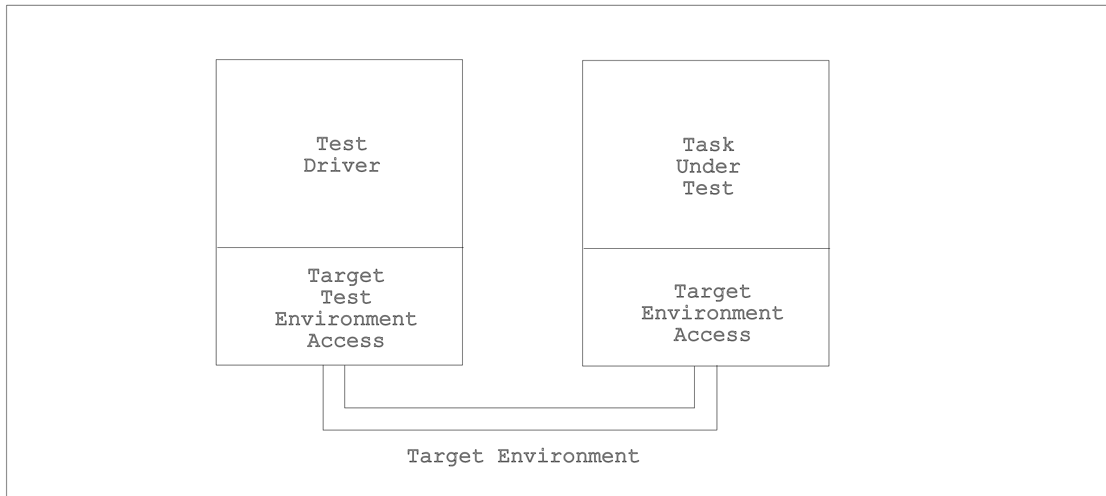

Figure 4.3 - Concurrent Test Structure (Target)

For cross-testing, the test driver has to be portable between the host and target environments. The target environment access and the target test environment access are both replaced with host simulations, allowing the test to be executed in the host environment, as shown in figure 4.4.

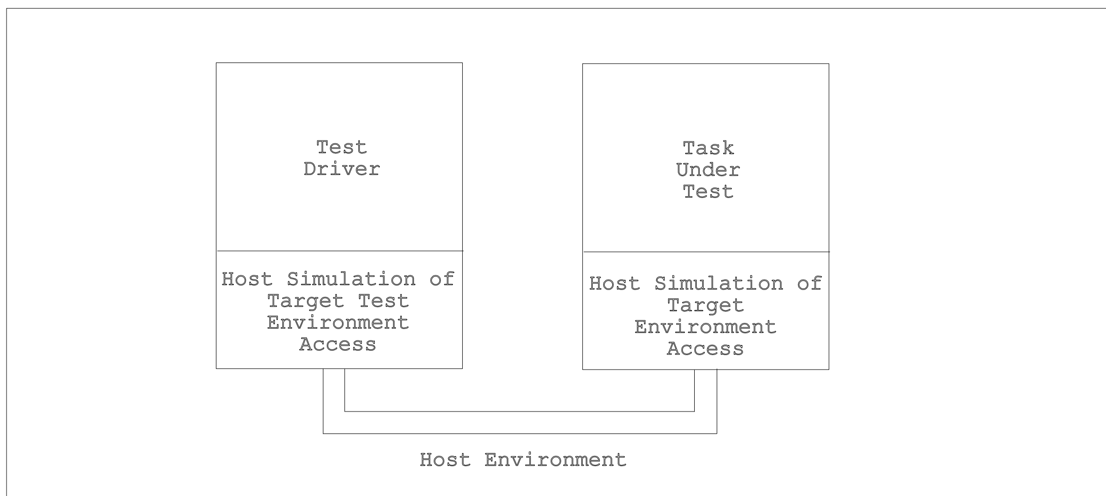

Figure 4.4 - Concurrent Test Structure (Host)

In practice, a cross-testing strategy would develop the simulation in the host environment first, allowing task tests to be developed and executed in the host environment, followed by confirmation tests in the target environment.

The host simulations of the target environment access and target test environment access form reusable components, which can be used again for the testing of other tasks. A similar architecture can be used to test groups of tasks during subsequent stages of integration testing.

Using the concurrent test structure for cross-testing requires detailed knowledge of both the host environment and the target environment. Test developers must beware of the dangers that small differences between the environments can make to the validity of cross-testing.

©IPL Information Processing Ltd

It is therefore important that the portability and validity of tests using a concurrent test structure developed for the host environment should be verified by confirmation testing in the target environment. If there is any doubt about the validity of cross-testing, it may be better to forgo testing in the host environment and only test in the target environment.

Real World Input and Output

At some point in the integration and testing of a software system, software which conducts real world input and output will have to be tested.

Such real world input and output may be to files, to communication devices, or to a user interface.

One way of testing real world input and output is to redirect it to the test driver. Many operating systems (both host and target) provide facilities for redirecting input and output. For example, on a UNIX system a `pipe' could be used to connect between software under test and a test driver. Other operating systems have similar facilities.

It is usually good practice to design software so that it is capable of redirecting input and output, even if such redirection will never be required in use. The benefits in ease of testing will soon offset any implementation costs.

This still leaves the actual physical input and output in the target environment to be tested. Figure 4.3 shows a test driver executing in the host environment being used to simulate inputs to and collect outputs from software under test in the target environment. The real target input and output devices are used.

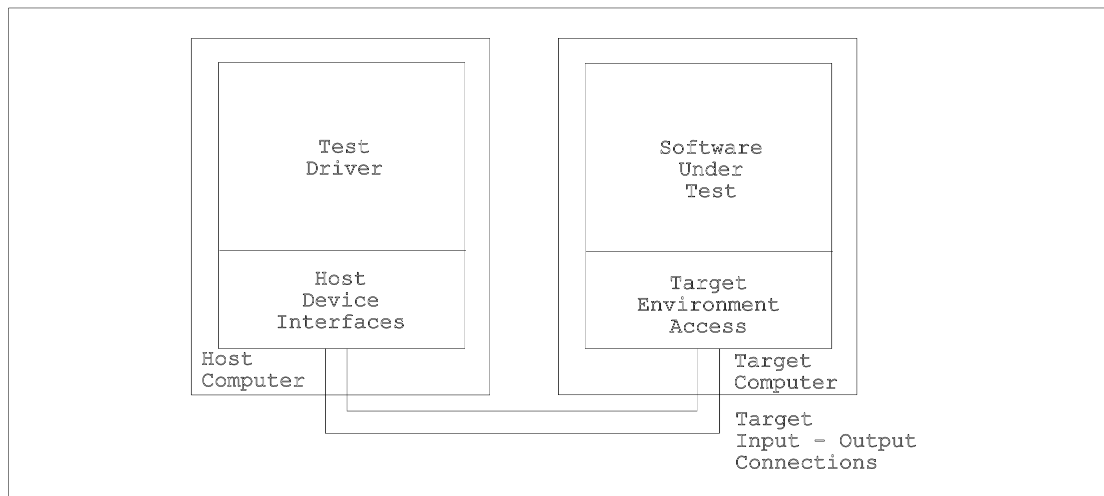

Figure 4.5 - Testing Real World Input - Output

Other techniques which can be used include the capture of displays and capture of GUI bit-maps. For example, a test driver could call a display capture utility which records a display or a GUI bit map to a file. The test driver can then call a file comparison utility to compare the captured file to a reference file. However, at this level of testing it is often better to use a special purpose tool.

A test driver can also prompt a test observer for observations, recording the observers response and comparing it to the required response.

©IPL Information Processing Ltd

Unpredictable Software

Testing may encounter problems of predictability. For example, consider some software which outputs messages in response to inputs. The behaviour of the software should be predictable and testing straightforward, with the test driver simulating an input and checking the response. However, suppose the messages contain a time and date field. The exact form of the outputs will be unpredictable, making it difficult to automatically check the correctness of test results.

Software can appear to behave unpredictably during testing when there are one or more factors which are outside of the control of the test driver. Test design has to anticipate problems of this nature and compensate for them.

5. Using AdaTEST and Cantata

AdaTEST and Cantata provide repeatability of tests and portability of test scripts between host and target environments. The stub simulation facility enables any unit, including target specific units, to be simulated in the host or target environments. More information on stub simulation can be found in the AdaTEST and Cantata manuals.

AdaTEST and Cantata can be used in both analysis mode and in non-analysis mode in both the host environment and in the target environment. A suggested strategy for cross-testing is:

a) Use the AdaTEST or Cantata instrumentor (in the host environment) to perform static analysis and to prepare an instrumented version of the software under test for dynamic (coverage) analysis.

b) Execute tests in analysis mode in the host environment, using instrumented software. Ensure that coverage objectives are achieved. Correct any errors in the software under test and in the test script.

c) Verify that tests execute correctly in the target environment by repeating tests in non-analysis mode in the target environment, using uninstrumented software.

Where target environment dependencies prevent a test from being executed in the host environment, (b) above should be executed in the target environment.

Where extreme levels of integrity are required, it may be advisable to precede step (c) with an execution of tests in analysis mode in the target environment, to verify that coverage is unchanged.

AdaTEST and Cantata can be interfaced to files, devices, and system utilities such as file comparison utilities in order to test real world input and output.

The CHECK_OBSERVATION facility enables a test script to prompt the test observer for a response as confirmation of correct function of the software under test. This can be useful when testing user interfaces. AdaTEST and Cantata can also be used in association with user interface capture-replay test tools, to control overall test execution, check results, and provide coverage analysis.

©IPL Information Processing Ltd

6. Conclusion

This paper has described a selection of techniques for cross-testing. All host-target software developments should be able to benefit from some of these techniques.

The efficiency of host-target software development can be maximised by an effective cross-testing strategy. This will usually involve performing the majority of testing in the host environment, only moving to the target environment to verify test results and for final system testing. Bottlenecks arising from insufficient access to the target system can be avoided, and investment in expensive target resources such as in-circuit emulators can be minimised.

An added benefit of such a strategy is that it provides a useful degree of flexibility when planning host-target developments. Should target hardware be late or unavailable for other reasons, verification of test results in the target environment can be postponed until the target hardware is available.

Designing software for portability between host and target environments is an essential prerequisite for successful cross-testing. The benefits of such a design do not end with cross-testing. Designing software for portability will generally improve quality and also provide benefits in simplified maintenance.

The portability of AdaTEST and Cantata test scripts between the host and target environments has been specifically designed to facilitate cross-testing, enabling tests to be fully developed in the host environment, and then simply repeated in the target environment. It is believed that AdaTEST and Cantata are the only tools to offer this functionality.

©IPL Information Processing Ltd

Annexe A Host - Target Communications

The question of how the host and target environments should be used to conduct testing depends greatly on the nature of the host and target environments, and the means of connecting between the host and target environments.

In figure A.1, the host is connected directly to the target. Software from the host is downloaded to the target, usually through a serial interface (a), or a LAN (b). A simple target debug and communication utility can be used to download and run target test software. Test results are communicated back to the host for storage.

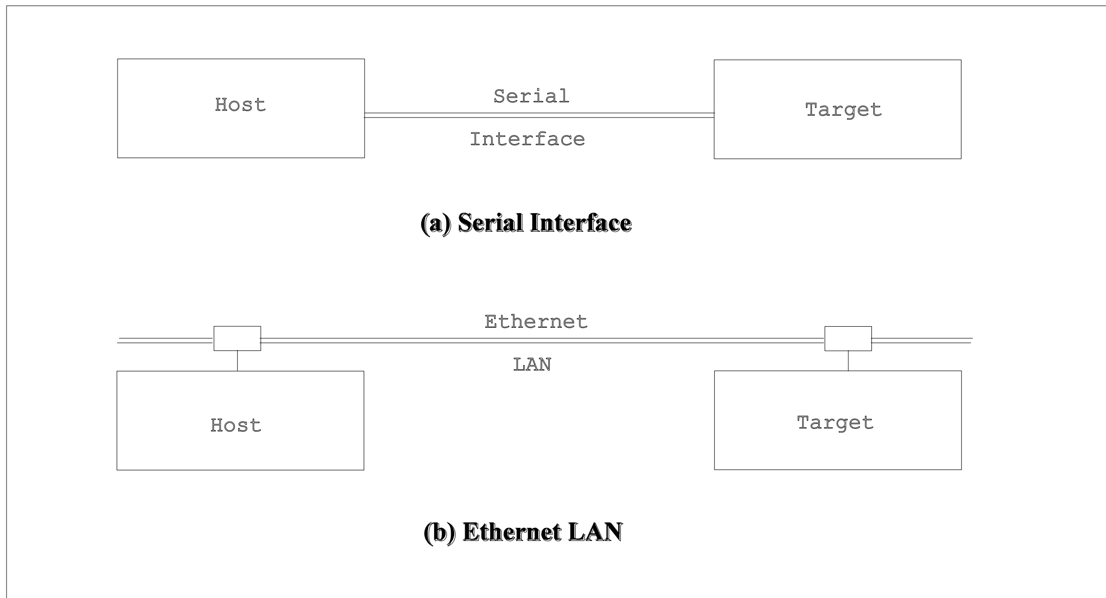

Figure A.1 - Direct Connection

The speed of communications is an important consideration when selecting host and target development environments. A LAN connection is usually a better choice, because the communications is faster and many hosts can easily share a single target. For example, suppose a particular test requires one Mega byte of program and data to be downloaded from the host environment to the target environment:

A serial link at 9600 baud would take a minimum of 17 minutes, without any allowance for communications overhead.

Using an Ethernet LAN, a similar download could be achieved in seconds, greatly increasing the effective time that the target environment is available for software testing.

Where a target environment does not support a direct connection with the host environment, or at levels of testing where a direct connection to the host environment is precluded, indirect connections have to be used. Figure A.2 shows the use of an emulator to connect to a target, with the interface from the host environment to the emulator being serial (a), LAN (b), or parallel (c).

©IPL Information Processing Ltd

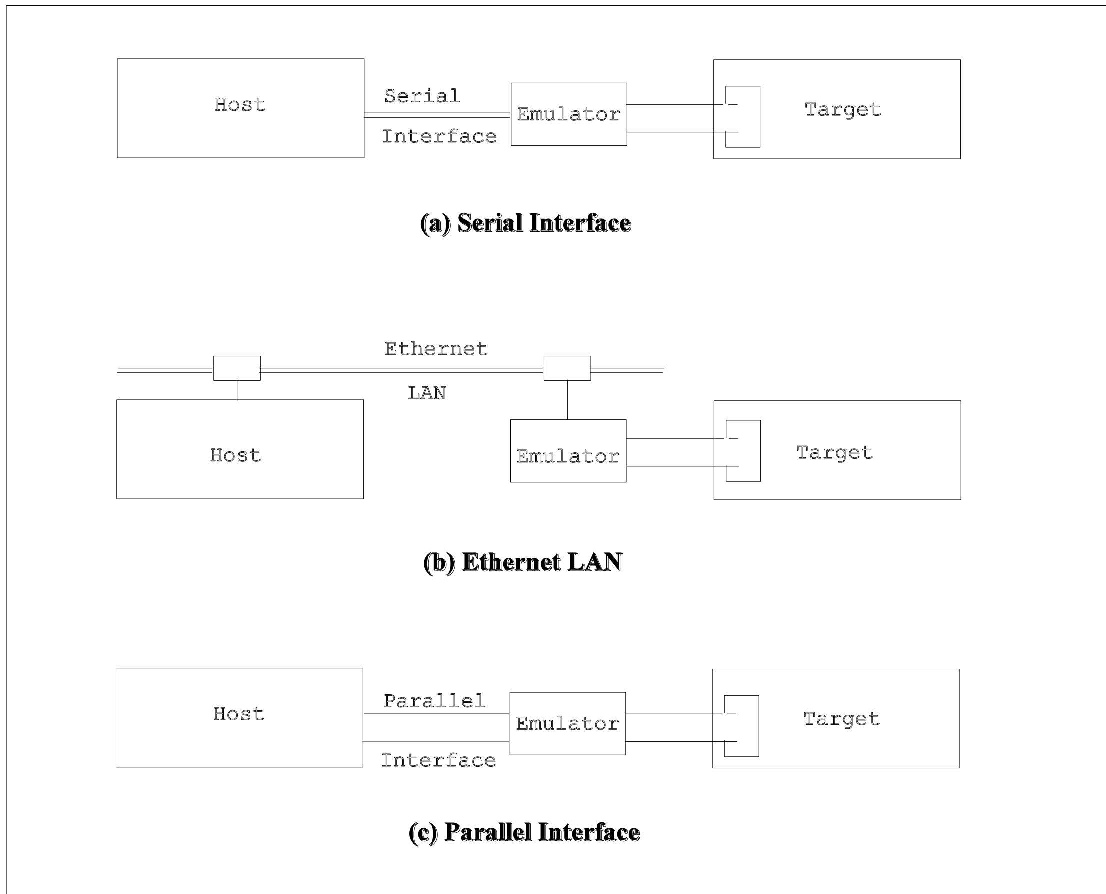

Figure A.2 - Using an Emulator

In general, the demand for high speed data transfer between the host and the emulator means that most emulators use LAN or parallel interfaces from the host. However, special high speed fibre optic serial interfaces are provided by some emulator suppliers.

Indirect connections also include the transfer of software using removable media such as floppy discs and tape cartridges, as shown in figure A.3. Removable media is often the method of choice when the target is a general purpose computer.

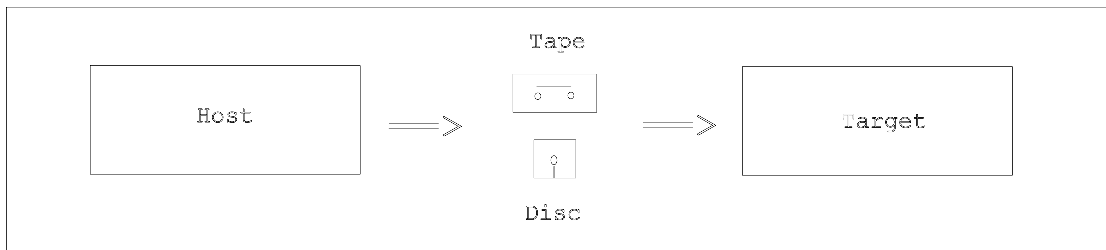

Figure A.3 - Indirect Connection using Removable Media

The use of Programmable Read Only Memory (PROM) to transfer software to the target, as shown in figure A.4, is usually the final stage of embedded system development. At this stage in the life cycle the main activity is system testing and acceptance testing, with real world input and output.

©IPL Information Processing Ltd

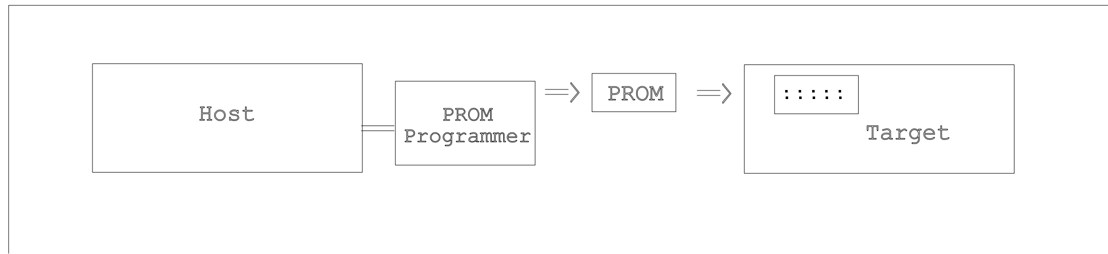

Figure A.4 - Software Transfer using PROM

The means of connecting between host and target will often vary between phases of the software life cycle. For example, consider the development of a real time embedded system:

Initially, an emulator might be used until direct host-target connection over a LAN is fully operational.

The LAN connection would then be used for all target work until the final stages of system testing.

Finally, software would be programmed into PROM and the direct host-target connection removed.

Another example, from a different area of software application, is the use of a dedicated host computer to develop software for a target computer which is already in continuous use by existing software. Under these circumstances, the only connection with the target might be a tape cartridge or floppy disc.

Target testing may require a display and keyboard to be attached to the target environment, either for user interface testing or for the presentation of test results, as shown in figure A.5(a). The host-target interface need not be a serial interface, a LAN could have been used.

Some embedded systems will not have a display or keyboard, or any means of attaching a display or keyboard should one be needed. Figure A.5(b) shows a display in the target environment being simulated by the host, with the associated display appearing in a window on the host display.

When available, such configurations of host-target connection provide two main advantages:

- The host environment can be used to log outputs made by a user interface and simulate inputs to a user interface.
- The developer has all host and target information available on a single workstation.

©IPL Information Processing Ltd

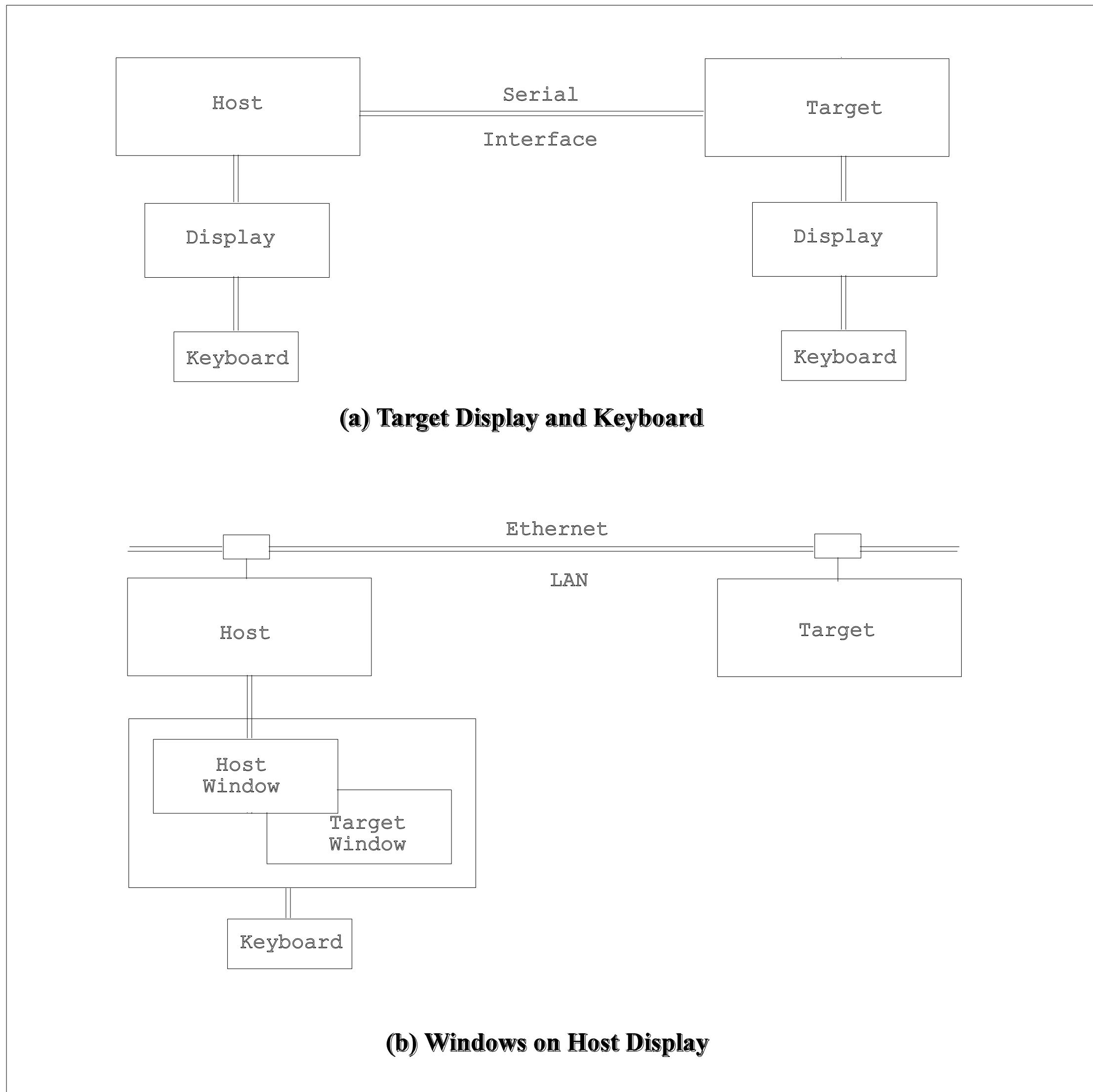

Figure A.5 - Target Display Options

Sometimes it will not be possible to use a single host-target interface for both downloading the target test software and for returning test results to the host. The simple solution is a second interface, as shown in figure A.6. The second interface need not be the same type of interface as the download interface. For example, an emulator could be used to download and run a target test program, with the results being returned to the host through a serial interface.

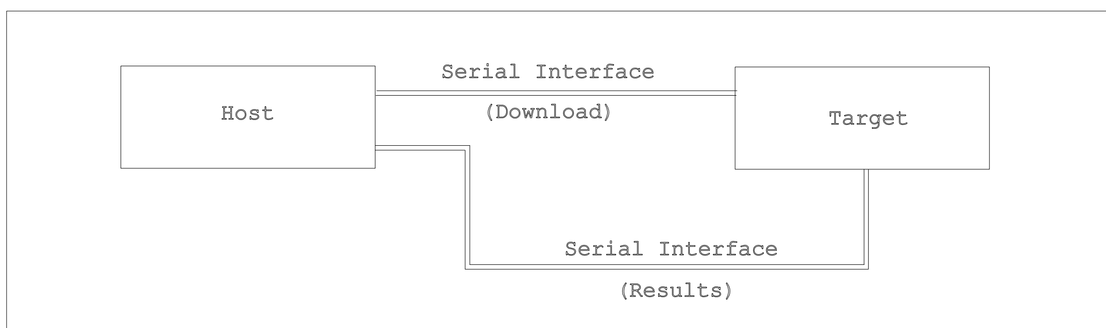

Figure A.6 - A Second Interface

©IPL Information Processing Ltd

There are many other means of connecting between host and target environments, involving other types of direct and indirect connections, and different combinations of the means of connection shown. Although other means of connection are not explicitly identified by this paper, the issues involved and the applicable techniques are similar to those described in this paper.

©IPL Information Processing Ltd