

# Software IP Reuse: Easing Integration and Test Pressure

Class # *Exhibition Showcase* at the Embedded Systems Conference, Farnborough 2009

By

David Whale

## **Abstract**

Continual growth in system size and complexity increases the need for software reuse. Recent research exposes many issues associated with successfully integrating and testing a large base of software, suggesting more could be done to energise the software IP supply chain to ease these pressures.

This paper exposes and explores some key integration and test pressures associated with reusing a large base of third-party software, providing attendees with practical advice on how to improve the success of complex integration projects.

At the end of this class, you should be able to:

- \* describe the complexity and issues of integrating 3<sup>rd</sup> party software;
- \* explain how and why an appropriate testing methodology helps;
- \* write your own requirements for tools to aid the process.

# 1. Introduction

As the complexity of software based products continue to increase, the need to reuse software in order to be competitive and meet tough market windows becomes more important. Adding third-party software IP to your product can often bring significant additional expertise and experience to your project, but it brings with it additional risks. Many of the problems associated with integrating complex third-party software IP into your product often only manifest themselves quite late in a development project when your project is already like a pressure cooker trying to meet a tough market window delivery deadline.

A number of advances in software methodologies, such as object orientation, have promised to give the software industry more of a plug and play approach to software construction, similar to that enjoyed by the hardware/silicon design industry. But, somehow, with software, the devil is always in the detail – it is still very hard to integrate large bases of software, and planning an integration project can still be quite unpredictable.

Involvement with a recent proof-of-concept research project into software IP reuse helped to identify some of the key issues and pressures with integrating third party IP, and this paper attempts to collect together both the results of this research, and a number of experiences from real integration projects, highlight the key pressure points in integration, and provide a suggested framework process that you can follow in order to pre-empt some of the pressures and manage the risk of future integration projects better.

# 2. What software might you reuse?

Consider an intelligent fridge magnet:

<b>Features</b>	<b>Reuse areas</b>
wired and wireless ethernet USB for memory stick use SD/MMC/CF touch screen high quality graphics flash hard disk zigbee home automation zigbee smart energy bluetooth/mobile phone connectivity power saving some form of operating system a set of applications various protocols wireless printing web browsing audio streaming	<b>operating system</b> linux? windows CE? others?  <b>applications</b> web browser, audio/video app, PIM  <b>peripherals</b> USB memory stick, iPod  <b>middleware</b> filing systems, printing, CDDDB, iTunes, audio, video and graphics codecs  <b>connectivity</b> ethernet, wifi, bluetooth, zigbee, USB  <b>protocols</b> TCP/IP, FTP, audio streaming, printing, HTTP

\* Are you an expert in each of these areas?

\* How many IP suppliers might you use to develop this product?

### 3. Where does the pressure to reuse come from?

Pressure to reuse comes from a number of sources:

**market heartbeat** – your competitors are getting to the market with new products and features quicker than ever. Last year's product is a bit outdated compared to the state of the market, you need your next generation product out on the market now.

**market window** – if you don't release your product in the next few months, your product will be outdated before it is released – i.e. you will "miss a heartbeat" and have to catch the next cycle, hoping you don't lose customer loyalty.

**limited development resources** – to keep up with the market heartbeat and market windows, you need to concurrently develop a pipeline of products, one planned for release while the next generation is being developed while the next generation is in the planning.

**payback of existing standards and infrastructures** – infrastructures such as connectivity (eg. TCP/IP) and common data services (e.g. iTunes, CDDDB), especially mature ones, can give your product access to a wider range of services and features that you could not hope to develop on your own in the time/cost available.

**User expectations** – these are higher than ever, and continue to grow, especially in the consumer market. Because the latest portable games console has wireless connectivity, users now expect all their other devices to work the same way – if you are a camera manufacturer you're now forced by user expectations to build wireless connectivity and social networking support into your next product.

#### 3.1. What will these pressures force you to do?

Diversity, complexity and growth driven by features, expectations, and standards means your next product will have lots of 3<sup>rd</sup> party software IP inside it. It probably won't all be from the same supplier, but it will all have to interoperate inside the same product, and it will be your job to make sure it works, all the time.

Your development teams will need to diversify and become expert in a range of different technology areas, and you will most likely work with a wider range of third party software IP suppliers as your products become more complex.

It should not be overlooked though, that once you engage with a third party software IP supplier, many products in your product line can benefit from the effort that you invest in integration, and there is much opportunity for multiple reuse across a whole range of products.

#### 3.2. What might you be reusing?

It is normal for those new to reuse, to assume that the major part being reused is software source code. However, the final software delivery is just the final outcome. Embedded into this software is a wealth of experience, and a good delivery may be accompanied by many other supporting items, such as:

code, design, architecture, design patterns, test cases, test frameworks, tools, techniques, experience of original developers, mileage of the product in the field, test conformance passes, pre-integrated solutions, managed risk, reference design, reference platform.
---

## 5. What problems will you have while integrating 3<sup>rd</sup> party IP?

### 5.1. IP Protection

It is only natural for a third party IP supplier, having invested heavily in developing a large base of software, to want to protect this software. Different suppliers will deal with this in different ways, and each has its own issues.

**Licence agreements** – The minimum protection you would expect from a supplier would be a licence agreement. This is the same for an open source delivery. Check the licence agreement very carefully, especially open source licence agreements, as there may be restrictions on what and how can be distributed to third parties, and obligations (especially with open source) to feed back fixes to any problems you find. If you find a potential processor/peripheral bug and have to send sample code to the processor manufacturer, you may not be allowed to send third party IP source under the terms of the licence, so you may be forced to extract the problem code into a small self contained test case.

**Source Obfuscation** – A full source code delivery is very flexible, but can be like the supplier giving away their crown jewels. Many suppliers apply source code obfuscation tools to their source before delivery, which provides a reasonable level of protection, and still gives you full control over compile time options, compiler optimisations and allow you to use any code instrumentation tools such as coverage checkers and boundary check tools, although some of the output of these tools can be as obfuscated as the source code you feed into it.

**Object libraries** – For very valuable IP, it is usual for suppliers to provide object code or object libraries. With this type of delivery comes a large collection of thorny problems. The main risk for you as an IP user is if the supplier goes out of business – it is normal for suppliers to set up an escrow agreement where in such a scenario, you get delivered the source code from an independent third party, but in situations where you get regular patches and bug fixes, you are at the mercy of the suppliers processes as to whether the escrowed source code is precisely the right version for your build of object libraries.

**Build responsibility** – The main issue with object code deliveries, is that the supplier is now completely responsible for the build of the code – which compiler version did they build it with and is it the same as the version you have used? When you upgrade your compiler, do they upgrade their compiler? Which compile time flags, compiler options and compiler optimisations did they build it with? How have they named and placed modules in the memory map and do you still have control with your linker to place critical code into high speed memory? How good is the supplier's process and will they manage your customer specific build options carefully for you when you ask for patches and bug fixes? Bug fixing becomes tricky, because again the supplier is responsible for choosing which bug fixes go into your libraries, not you.

**Configuration control responsibility** – The supplier is completely responsible for configuration control of these object libraries – if you have multiple products in the field with different versions of the object libraries, can the supplier trace each library back to the original source and the original build configuration and correct compiler version in order to reproduce it for patching and bug fixing? Some suppliers may charge more in order to manage separate object library configurations.

**Debugging** – When code goes wrong and you step into these object libraries with a debugger or emulator, there won't be any symbolic information, so the code will be a total black box and pretty impossible for you to find memory leaks and memory corruptions.

## 5.2. Documentation

**Auto documentation** – By far one of the biggest obstacles at the start of an integration project, will be the quality (or lack of) supplied documentation. With the popularity of auto documentation generators such as doxygen and javadoc, it is normal for API documentation to be auto generated from tags in the source code. However, for large complex multifunctional code deliveries, API documentation may give you several thousand functions to call, but no clear picture of how to tie these functions together to make the software do something useful.

The main issues with using API documentation tend to be:

**Valid sequences** – what sequences of API calls make something useful happen?

**Parameter setup** – For API calls with complex parameters, what exactly do you need to pass in to configure the call correctly?

**Industry terminology** – Where there is a significant amount of industry specific terminology, this terminology will heavily leak through to API specifications – how can you make sure you understand enough of this terminology without becoming an expert?

**Open Source** – For open source deliveries, there is much evidence to show that because the source code is completely available for inspection and modification, there tends to be less supporting documentation (the hope being that the source is self explanatory). For a large complex base of software, it may take some considerable time (especially in object oriented systems with deeply nested inheritance hierarchies) for you to understand enough of the source in order to answer your present line of questioning.

## 5.3. Version Control

**Library versions** – Throughout the course of a complex integration project, depending on it's maturity, it is likely you will receive many deliveries of your third party IP. Without proper controls over these deliveries, it is very easy to find developers on a team using the wrong version or wrong configuration of a supplied library or batch of source. What version of the third party IP library did developer X test with, before checking their code into your version control system? Is this the same version that developer Y uses when taking a copy of developer X's changes?

**Sets of files** – Loosing track of the relationship between multiple files in a 3<sup>rd</sup> party delivery can cause some very obscure build or runtime errors, so it is vital that all files (along with a suitable release description) are updated as a set. Checking in untested code to the repository could have a large knock-on effect to all developers down the line, so the ability to have a simple "build and smoke test" for 3<sup>rd</sup> party IP that is run before checking into the repository can avoid these problems.

**Deprecation of API calls** – 3<sup>rd</sup> party IP suppliers will always have to enhance and develop their solutions, and over time it is normal for API calls and features to be deprecated and replaced with updated versions and methods – this means that the API documentation that accompanies a particular release is part of the set of files that change with a release – it is advisable to version control these files along with the code delivery, so that developers can always access the correct version of the API documentation.

**Multiple concurrent library versions** – If you make good use of reusable IP, then it is probable that you will have multiple products built around the same piece of 3<sup>rd</sup> party IP – but beware that older products may be based on older API versions, so you will need some method to be able to manage multiple concurrent versions or variants of 3<sup>rd</sup> party IP inside your version control system.

**Configuration builder tools** – Complex IP deliveries have many configuration options, and it is normal for suppliers to provide application or configuration builders – usually windows GUI programs, that manage all the options and flags for you, and auto generate either an application framework or a set of configuration files. There can be complex rebuild issues further down the line if you don't carefully control the tools, the input files and the output files as a set. To reconfigure and regenerate a new configuration, you will need the correct version of the builder tool, and the correct input files to the tool. Your build system will need the output from this process – but it is usual to put some form of version control over all three items, and this becomes especially important as products move into their maintenance phase and as IP suppliers release newer versions of tools for your newer products to use.

## 5.4. Platform differences

Many thorny problems can occur where there are significant differences between your processor platform and the platform the IP was developed and tested on. If you have a source code delivery, you can fix many of these yourself without involving the IP supplier.

**Processor architecture** – a different endianness may be important. Good deliveries will use or provide macros and compile flags to change the endianness, but you will need to configure and check this.

**Shared libraries** – not only do you need to check you have the correct compiler and options, but you should check you use the same/correct runtime libraries and keep these under some form of version control, and all your development team must be using the same version of the runtime libraries (otherwise you are not performing a valid test). Bugs caused by the wrong version of a shared library can be very hard to track down, as they may only manifest themselves inside certain developers builds.

**Limited architectures** – if porting to a processor with a harvard architecture (separate program and data busses), you may have to configure separate addressing modes for data read from flash compared to data read from RAM, and on some processor architectures and compilers you may get incorrect code if this is not correctly configured. Take special care to look out for use of library functions like memcpy and strcpy, which on limited architectures may have different versions depending on which memory space objects are located in.

**Compiler certification** – some IP suppliers certify their software against a specific compiler and version. While this does not mean it will not work on other compilers, you may loose out on supplier testing if you don't use the recommended version. This issue is additionally thorny if you have multiple pieces of IP from multiple suppliers, each with different requirements. Sometimes it may be necessary to push back on suppliers to get them to certify their IP with later compiler versions.

**Reference platform and reference design** – moving between a supplier supported platform and your platform can introduce many complex differences and problems, and in this case, the better IP suppliers provide a reference design or reference platform. The

reference platform is a very powerful test tool. It is not only useful for initial evaluation, but it can form a vital component in the bug fixing cycle once you have ported to your specific platform. Always aim to keep the reference platform working and use it extensively in your testing where possible, as this will provide a common platform to work on when communicating with your IP supplier (especially if they don't have access to your platform).

## 5.5. Configurations

With any large base of software IP, for an embedded system, footprint control will be vital – if you cannot fit all the required IP inside the resources of your chosen processor platform, you don't have a product. Many IP suppliers provide some control over the footprint of their deliveries.

**Optionality** – The most common footprint control mechanism is where an IP supplier makes whole sets of features optional – typically by not including specific files in the build (for object deliveries) or setting compile flags (for source deliveries). Check very carefully that you can trim your footprint down sufficiently with the pre-determined optionality provided by the supplier – optionality is a form of dependency control, and the supplier would have only tested certain combinations of optional components. If you try to build a configuration that has not been tested by the supplier, you will often be blazing a trail into the unknown and may lose support from your supplier.

**Configurable libraries** – object code deliveries will often provide some limited configuration options – normally done via callbacks or linkable functions. A callback will be registered at runtime when starting the library, whereas a linkable function will normally be provided as a stub that you can modify. Stubs and callbacks should be written very carefully, as often they are called at critical moments in the flow of the library software, and obscure and hard to debug problems can be introduced by spending too long in the callback or incorrectly changing the state of the library. Sometimes an asynchronous mechanism (e.g. setting a flag to be processed later) is required to limit the time spent in a callback.

## 5.6. System wide issues

**Who is master** – There appear to be two camps of IP suppliers – those who assume their application has full control of the system (and you build your application inside theirs), and those that assume they are a module in a bigger system. Problems tend to occur when you have multiple IP libraries from multiple suppliers – really you want your application to be the master in this scenario, otherwise you may be forced to make some hard choices as to how some of the other items listed below are dealt with.

**Concurrent access to CPU** – depending on the size of your system, you may or may not have an RTOS. The IP that you purchase may or may not be written assuming an RTOS is present (and may or may not assume a particular RTOS is in use). Multiple IP deliveries may have different requirements for managing threading and tasking, using semaphores and events and other communication mechanisms. Check very carefully what the requirements are for any IP you purchase, and check what is available on your chosen platform. Typically IP that provides one or more "application tick hooks" can be quite easily made to run on both RTOS and non-RTOS systems – but those that are designed with an RTOS in mind can be much harder to run on a non-RTOS system. Scheduling access to the CPU is a system wide issue, and your system needs to be architected as such.

**Power saving** – especially important on battery powered consumer devices, power saving is a system wide issue that can be affected by all modules in a system. An errant piece of IP that prevents the CPU from entering a sleep mode can reduce battery life significantly.

To squeeze power saving to its limits is a complex system wide issue, and any third party IP must have the necessary interfaces to allow devices to be entered into sleep or low power mode (e.g. turning off a radio power amp), and for the state of the software to be informed of the intention to enter sleep mode.

**Shared busses** – e.g. SPI or I2C busses in your design may need multiple devices connected to them – an Ethernet MAC/PHY on SPI along with an LCD display and a serial Flash device for example. Third party IP may be written assuming that access to devices is exclusive, but on a shared bus, especially with interrupt handling and multiple devices, it's up to you as the system architect to ensure that devices are accessed at the right time, without locking out other devices, and without conflicts or deadlock occurring.

**Initialisation** – all IP requires correct initialisation, and there can often be complex startup dependencies where certain hardware or software has to be initialised in a certain order. Typically, handling initialisation with a state machine, and splitting driver/hardware initialisation from software state initialisation can be a way to break complex dependencies, but make sure that your third party IP has enough control over initialisation, especially where shared busses or multiple third party IP modules are in use, as some complex and hard to debug race conditions can occur.

**Watchdog timeouts** – long running operations inside third party IP can easily cause your watchdog timer to bite, especially if you don't know what the worst case execution paths are. Some IP suppliers provide hooks or macros that you can define that feed the watchdog at key points, but the problem gets harder with object code deliveries where macros cannot be used.

**Runtime library trust** – it is common for third party IP to have different levels of trust against runtime libraries and other reusable libraries. In one system worked on by the author that had IP from multiple suppliers, it was discovered that the system had 13 different copies of `strlen()` and 3 different XML parsers. The `strlen()` situation occurred because each IP supplier wanted a specific guaranteed operation and performance (e.g. taking advantage of high performance assembler compare instructions, handling unicode correctly, etc). The XML situation occurred because each IP supplier independently chose a third party XML parser that met their needs. If memory footprint control is vital, one course of action here is to push back on the IP suppliers to build functionality behind a common defined interface and share code – but this will put additional pressure on the testing and some suppliers may not certify their solution for example if you use a different XML library.

**Base data types** – each supplier tends to have their own standards for definitions of base data types (e.g. `Int16`, `Int32`), and this can cause conflicts when including IP from multiple suppliers, often causing build problems where there are multiple or different definitions. It is sometimes necessary to move these data type definitions out from third party IP files into a common definitions header file, but beware that this introduces a necessary "re-integration" step every time you get a new delivery from the supplier.

**CPU loading** – each IP delivery will have it's own requirements for access to the CPU, and the profile of execution will differ both based on the IP and the mode of that IP. It is up to you as the system architect to ensure that enough execution time is given to each block of IP at the right times – a correctly configured RTOS with correct process priorities can make this possible – so can a carefully balanced cooperative round-robin top loop. The key issue here is that if third party IP is starved of CPU time at the wrong time, very hard to debug problems can be introduced. It is always worth considering some form of code

instrumentation that either measures execution time or makes it visible on a digital IO pin, so that when this situation occurs it is obvious. Putting code asserts around code that fails to execute to a deadline can also help track down these types of problems.

## 5.8. Building the system

A complex system with IP supplied by 6 different IP suppliers, which is ported to your own custom platform, needs to be built using predictable methods so that the code generated can be trusted to be correct, always.

Each supplier will almost definitely choose their own build system – perhaps one that comes with a specific toolchain or IDE, and perhaps with some custom build steps where complex configuration is required. There is some evidence of more suppliers choosing an eclipse based environment to deliver their IP, but this is not exhaustive throughout the industry – so at some time you will be faced with IP from multiple suppliers with multiple methods of building it.

Deliveries of new code from IP suppliers will have been built and tested using the build tools chosen by that supplier, and you will have to re-integrate their changes into your chosen build system.

There are three main methods of simplifying this problem (which introduce their own specific sub-problems):

**Conversion** – get your toolsmith to write a conversion script, that takes build scripts supplied by your IP suppliers, and auto convert them into your chosen build scripting environment. These scripts can be hard to write if the supplied build scripts are in a binary format or have complex dependency rules (e.g. make files), but it does give you a single unified build system.

**Wrapping** – embrace and use the multiple build systems from your multiple IP suppliers, and make the process of building third party code a sub-step of your build process, using the tools provided or chosen by each IP supplier. This does however require every developer to install and use all tools from all IP suppliers, along with the associated version management issues.

**Separate build** – make someone in your organisation responsible for accepting new third party IP releases and building/testing the code, and then releasing prebuilt and preconfigured libraries and header file sets into your version control system for all other developers to use. This can introduce a bottleneck into the process of adopting new third party IP (which could occur for every patch and bug fix provided by the supplier), but it does simplify the build process where you have a large number of developers relying on the same third party IP.

## 5.9. Testing

Testing and re-testing large blocks of 3<sup>rd</sup> party IP can be both time consuming and complex. Part of this is due to the IP being a black box, and part of it will be due to your unfamiliarity with the specific industry area (especially for very large and complex IP such as GSM/GPRS).

**Applying patches** – there will be times that IP suppliers will provide patches or bug fixes, and you will need to re-test them inside your system. Where you have a large number of developers all building code, introducing a poorly tested patch can introduce many days of down-time for your developers. It is helpful on certain team sizes to have one developer

dedicated to accepting and testing and re-integrating 3<sup>rd</sup> party IP patches, so that what is later picked up by all other developers is known to work inside your product. Remember that patches applied to 3<sup>rd</sup> party IP will often only be tested on the reference platform – if you have a source code delivery and if your source code has been customised to make it work on your platform, each patch supplied by the IP supplier will have to be reviewed and re-integrated onto your platform, so good version control tools and good diff-ing and source code merging tools are vital.

**Executable tests** – some standards based IP may provide external executable test harnesses – USB for example has the USBCV tool that can be applied to any USB product to verify correct command operation. Other standards based IP may have similar mechanisms, but sometimes it is necessary for your product to have necessary test interfaces or test modes in order to use such tests.

**Test scripts** – for very complex standards based IP, there may be a set of test scripts and/or industry standard test tools (e.g. in GSM/GPRS). These will form part of a bigger regression test suite that you will need to write for your product. Testing can become very complex and time consuming where there are multiple methods of testing each individual block of IP, and on large systems this can be an area where your toolsmith can help to try to automate the various forms of testing. You might consider purchasing an off the shelf configurable test framework that can be interfaced to all the various test systems, but don't underestimate the amount of time required to build an effective test platform.

**The reference platform** – once again, the reference platform is a vital piece of test equipment. IP suppliers will usually not have visibility of your platform and product, and therefore they can only perform and validate tests against their supplied reference platform or reference application. When you report potential bugs, they will want to see them reproduced on the reference platform to factor out any of your specific platform issues. When they deliver bug fixes, they will have tested them on the reference platform. It is vital that you keep and use the reference platform as an integral part of the bug fixing cycle, and that you have some way of building your configuration of the 3<sup>rd</sup> party IP to run on the reference platform, as this will significantly speed up the bug fix cycle.

**Multiple test tools** – with a product that has 3<sup>rd</sup> party IP from multiple suppliers, it is probable that you will have multiple reference platforms, multiple test scripts, multiple deliveries, multiple testing methods, and multiple test plans to run through, further complicating the testing process. Having a local person who is a nominated expert on a particular 3<sup>rd</sup> party IP delivery and tools can ease pressures in this area, but can create a bottleneck, so their time will need to be carefully managed.

## 5.10. Debugging

**Object code debugging** – it can be very hard to debug an object code library, stepping into code in a debugger or emulator without symbolic information will make it very hard to understand what the code is doing and how it works. The better IP suppliers will provide a good set of status return codes, and perhaps some form of embedded debug code, to aid in the debugging process. One of the hardest to debug problems will be a pointer dereference inside the object code that causes memory corruption – this could be due to a bug inside the object code, or it could be due to invalid parameters passed into the object code library – both have the same effect. Check if your IP supplier has included any form of embedded debug or mode tracing and turn it on to aid debugging.

**Knowing what the code should be doing** – Sometimes with 3<sup>rd</sup> party IP, it is not only the fact that the code crashes inside an object library that is hard to track down, but it is harder

if you do not know what the code should have been doing at that time. With complex and large 3<sup>rd</sup> party IP, having a good set of "smoke test" test vectors or use-cases can help to track down problems. Sending in test vectors (perhaps even on the reference platform) and comparing the response against some documented known response can aid developers in understanding what the code should be doing, before trying to debug a specific failure case.

**Different context of use** – often IP is reused in a different context of use to where it was developed. The author was involved in debugging a speech recognition engine that was originally designed for a PC, but was being ported to a mobile phone handset. In this particular setup, the 3<sup>rd</sup> party IP developer did not have access to any of the debug tools they were used to, they were not familiar with the platform, and the audio stream was sourced from different (GSM) codecs. Progress was not made until a representative set of test and visibility tools were created that presented input and output data to the 3<sup>rd</sup> party IP developers in a manner they were familiar with. Sometimes it is necessary to ensure that any debug expected by 3<sup>rd</sup> party IP suppliers can in some way be "tunnelled" out through your proprietary test interfaces, so that your suppliers can work in a familiar environment.

## 5.11. Global teams

It is becoming more common for teams to be spread geographically, across timezones, especially where a large collection of 3<sup>rd</sup> party IP is being integrated. A typical team for a large product development may involve many developers both within your organisation and outside of it in many geographic locations, and this hinders communications and progress.

**Timezones** – timezone differences between developers introduce long communication delays and can make debugging and development reminiscent of the 1960's batch programming. Social networking tools such as wiki's and forums and email have been shown to work well here, ensuring that all stakeholders on a team have a consistent view of the problems and issues.

**Getting the team together** – it is not always practical, but the communications bottlenecks of multiple timezones and multiple locations can sometimes be eased by having planned integration days, where groups of developers meet and deal with a pre-prepared list of issues, and such "integration days", if carefully planned to occur at the right time, can significantly accelerate problems and solve a number of issues in one hit.

**Development and test tools** – different teams will have different development and test tools, and this can make reproducing tests slow and difficult – sometimes all that is required is for two developers to agree that they are observing the same problem, but if developer X cannot send through a reproducible test case to developer Y due to different build tools, different test tools, or a different platform, diagnosing and investigating issues can become very slow and tedious. Finding some way to share test cases between developers can speed up issue resolution – e.g. having a common logging tool that can monitor and capture runs from a system, and be read by both parties, to give both parties a common language to converse and hypothesise about issues.

**licensing restrictions** – sometimes it is impossible for licensing reasons to share source code with other 3<sup>rd</sup> party developers, and this additionally slows the issue resolution process. Having "integration days" and using "common logging tools" can ease this situation.

## 6. How could you plan your next project to be a success?

There are three main things you can do to ease integration and test pressures:

- 1) Plan an *integration* project, not a *development* project.
- 2) Use a testing methodology based on an integration plan
- 3) Push back on your suppliers to provide better IP deliveries to ease the integration

### 6.1. Plan an integration project

The complexities and problems discussed in the earlier sections of this paper, should highlight that integrating 3<sup>rd</sup> party IP (especially from multiple suppliers) can become a hugely complex project, because there are many items that are out of your control compared with writing code in-house.

This requires a mindset change on your behalf – to recognise that it will be these additional integration complexities that can cause your project to fail, and that you should plan to manage these risks very carefully. Your internal development team merely becomes just another 3<sup>rd</sup> party in this integration project (but one you have a bit more control over, perhaps).

#### Reconsider the following:

- what people do you need and what skills do they need?
- what additional processes are required?
- what tools do you need?
- what are the major risks, and how can you avoid them?
- who is the named person technically responsible for making everything work together?

### 6.2. A model and testing methodology to help us talk about 3<sup>rd</sup> party IP selection

- usually driven by the product manager
- looking for a range of features, both technical and non technical
- trying to get a good price point for the features
- licensing issues dealt with here

#### evaluation

- usually driven by the product architect
- a very technically driven phase
- basically working out how[if?] we can make it work in our product
- might consist of a 'trial port' to show something working

#### integration

- usually driven by the platform engineer
- many of the problems happen at this phase
- this is where the devil in the detail appears

#### deployment

- usually driven by a test engineer
- testing, re-testing, delivering, bug fixing

### **6.2.1. What should you ask at the selection phase?**

- how do I find and select the right product
- how do I decide which is the best solution
- can I see the long term future
- can I see it enabling me to become successful
- how can I assess the cost/quality balance
- how can I assess short term need vs long term flexibility
- how can I see if it matches our required feature set
- can I see it supporting my product roadmap
- will my team be able to evaluate it to enough depth?
- will my team need to become experts in order to use it?
- what are the hidden costs?
- what is the licensing/cost model and is it compatible?
- are the supplier pushing the IP in the right direction for us?

### **6.2.2. What should you ask at the evaluation phase?**

- how will I understand it and configure it?
- how will I design an integration with our system?
- how can I assess the size of any porting job?
- how can I assess cost to port integrate and deploy?
- how can I assess complexity/time of hardware integration?
- how can I work out profile of the team I need to port/integrate?
- what additional training will the team need?
- what areas will my team need to become experts in?
- how expert will my team need to become in order to make it work?
- how can I fit this into my dayjob?
- how can I validate my evaluation answers
- how to build it
- how to test it
- how to modify, integrate and customise it
- what bits need to be provided to complete it
- measuring performance
- very much later, how it works – don't really need to know?
- key touch points
- key variance points
- packaging of documentation

### **6.2.3. What should you ask at the integration phase?**

- does it work out of the box and does it do something credible?
- how far away is that from what I want to do with it?
- how do I work out if it is doing the right thing (always)?
- when it goes wrong, how do I work out why and how to fix it?
- how do I know what is required to port/integrate to my platform?
- how do I measure and improve reliability?
- how do I know which gaps to fill in?
- how do I reproduce and report bugs and get them fixed?
- how do I integrate supplier provided bug fixes on my platform?
- how do I integrate supplier product upgrades on my platform?
- how do I work out the best way to fit it into our architecture?
- how can I estimate how long it will take to integrate?
- how will I cope with multiple IP's from multiple companies?
- who is your toolsmith that helps you automate things?

#### 6.2.4. What should you ask at the test phase?

- how do I work out what it should be doing?
- how do I work out if it is doing what it should be doing?
- how can I identify and fix bugs?
- how can I reliably automate testing?
- how can I detect a pass/fail on a test?
- does the supplier do anything to help me fix bugs?
- how can I locate the bug as core code or our integration code?
- how do I integrate and test bug fixes from supplier?
- how long will it take the supplier to fix bugs?
- embrace and use the reference platform in both directions
- automate test suites
- have a toolsmith
- test driven integration projects
- embed/wrap supplier test suites into your test suite
- build a regression test suite and automate it
- diffing tools (how has supplier code changed from last release)
- perf tools (where are the key bottlenecks, have they moved?)
- test tools (embed supplier scripts with your scripts)
- use the reference platform and reference implementations as your friend

#### 6.3. Could tools help?

Here are some ideas of how tools (either provided by the supplier or written by your toolsmith) could ease various pressures:

**selection** – perhaps better performance reporting, especially important if you are working out how to get enough MIPS out of your platform with various other IP integrated onto it.

**evaluation** – the ability to run test scripts from the supplier and run through and observe/monitor challenging use cases will force you to engage with the IP and understand how it can be made to work on your platform. Large IP will have a large base of documentation, so better documentation and better search may help. Complex IP with lots of options may benefit from more featured application builders that build you a working framework, but beware of the “who is master” issue.

**integration** – reproducing bugs and having conversations with suppliers is always hard, perhaps some form of common logging format that can be used to capture traces from your platform and talk intelligently with the supplier. Tools to log and tunnel supplier generated traces through your debug/trace system will be vital here. Building large IP deliveries inside your platform build always throws up complex issues, so more work on build tools could help to ease the build pain. Timezone differences when communicating with third parties often introduce large delays, so perhaps more use of social networking and web based collaboration tools will ensure everyone has a single common picture of the issues.

**deployment** – having taken the IP out of it’s original context of use (the reference platform or reference application), being able to run side-by-side tests on the reference platform and on your platform, and building this testing into any regression testing framework will speed up bug detection and fixing. Even with a large base of test cases, a large base of IP will be complex to observe – some form of “normality checking” tools could be developed to identify valid and invalid sequences.

## 7. What might you do differently?

### 7.1 IP users

- plan an integration project not a development project (test driven)
- embrace reference platforms and use them extensively
- build automated test harnesses
- build supplier tests into your tests
- have a toolsmith on the team

### 7.2. IP Suppliers

- deliver more appropriate documentation (task based)
- key observation points for debug
- tunneled debug
- tunneled test modes
- better performance characterisation
- dont assume you are master in the system
- dont assume you are the only 3<sup>rd</sup> party IP in the design
- build and test scripts packaged as reusable in different contexts
- provide debug that can be included in target and tunnelled through customers own debug/trace mechanism – don't assume you are master and don't assume printf is good enough.
- reference platforms or reference implementations
- reference applications – they will be used as a basis for an app, so write well with clean integration and configuration.
- software firewalls – protect API's to the hills.
- need something more like a cookbook – a collection of reference applications and sample code sequences showing how to make the sea of functions do something task oriented and relevant to the programmer.
- challenging use cases
- locating relevant documentation of large base of IP – most documentation written to assume you understand how it all works. As an IP integrator, you don't know how it all works so you don't have that context.

## 8. What should you take away from this paper?

### 8.1. IP reusers

- plan an integration project, not a development project. Use the test driven methodology.
- have a shopping list, and select a supplier that understands how to package IP
- build a regression test harness
- look very closely at your concurrency & resource sharing architecture

### 8.2. IP suppliers

- don't assume you are master or the only piece of 3<sup>rd</sup> party IP in the design
- task based documentation
- good reference implementations
- reusable test scripts