# Intelligent Design Environment:

# The key to rapid wireless embedded system development

*Developers are increasingly incorporating additional software such as wireless protocol stacks into their embedded products creating large, complex systems. Building and debugging such complicated systems consumes valuable resources and extends project timescales. The use of an Intelligent Design Environment eases the challenges and allows engineers to focus on their end application*

By removing the limitations imposed by cables, wireless technology has become a crucial aspect of our daily lives. We are only just beginning to explore the range of possible applications, yet already wireless technology is becoming pervasive by being built into everything from wearable electronics, smart phones and tablet computers, to refrigerators, cars and vending machines.

The potential of short-range wireless technology can lead to large profits for products boasting innovative embedded applications. (In this article, “wireless technology” is defined as communication without cables, whether based on radio frequency (RF) or another part of the electromagnetic spectrum. “Short range” is defined as communication up to 200 metres, but typically less than 20 metres.) However, wireless design is not trivial. There is a huge learning curve to overcome to understand the numerous technologies, their features, versions, components and limitations. And even after a technology has been selected the challenge becomes the construction of an application layer to utilise the many features within the technology in an optimal way that matches the specific requirements of the end product.

But with the support of a software interface known as an “Intelligent Design Environment” engineers with embedded software skills but limited or even zero RF design expertise can experience trouble-free planning, building and testing of their wireless-equipped solution. By using an Intelligent Design Environment and leveraging vendor expertise, the developer can merge the wireless technology’s protocol stack with his or her wireless application code, debug the system and quickly move on to complete the embedded software package required for a commercially-successful end product.

## *Growth and impact of wireless technology*

The wireless technology sector is already large yet growing rapidly. *The Economist* magazine has likened today’s wireless technology applications to the early days of electricity. At first, it was envisaged that electricity would be used solely to power incandescent bulbs; it wasn’t until the invention of the wall socket that the true potential of this physical phenomenon was realised. We are at that stage with wireless technology today; despite the thousands of innovative applications that already make use of RF connectivity we have only realised a fraction of its true potential.

But things are moving quickly. For example, over five million Bluetooth chips shipped every day throughout 2012, adding up to two billion units in that year alone, according to analyst InStat. And consulting company IHS forecasts that between 2011 and 2017, unit shipments will grow from 1.6 billion to 3.1 billion a year.

The latest release of Bluetooth, version 4.1, includes Bluetooth low energy (BLE). As a low-power variant of the technology, BLE extends wireless-connectivity capabilities to thousands of new products as the low power consumption enables devices to run from coin cell batteries. This new variant will move Bluetooth beyond mobile phones and other consumer electronics, into personal area network (PAN) applications such as heart rate- and speed & distance-monitors.

Wi-Fi, ZigBee, ANT and a slew of proprietary technologies have carved their own successful niches connecting hundreds of millions of products to the "wireless ecosystem". Wi-Fi, for example, has established itself as the standard for consumer-electronics wireless access to the Internet and is now routinely built into smartphones and tablet computers. According to telecommunications giant Cisco, Wi-Fi will consume more bandwidth than wired devices by 2015, sending 37.2 EB (1 EB is equal to 1 billion GB) of data wirelessly compared to the 37.0 EB of wired data that will be transmitted.

## *Using a multitasking approach for wireless embedded devices*

There's no escaping the fact that the underlying software architecture of a wireless device is complex. A typical architecture incorporates a number of embedded software components.

A popular strategy for software development for complex embedded systems is multitasking. There are two ways to manage multitasking: Pre-emptive scheduling and co-operative scheduling. Whilst the multitasking strategy can be used for any embedded development it is of particular relevance to wireless product design. Multitasking is advantageous for wireless applications because it avoids delays in communications whilst allowing for efficient use of processing resources.

The difference between the two techniques is that during pre-emptive scheduling the operating system (OS) determines when a task relinquishes control of the processor, whereas during co-operative scheduling, the task itself determines when it relinquishes control of the processor. Each has its advantages and disadvantages. (See Appendix A "*Pre-emptive scheduling vs. co-operative scheduling*".)

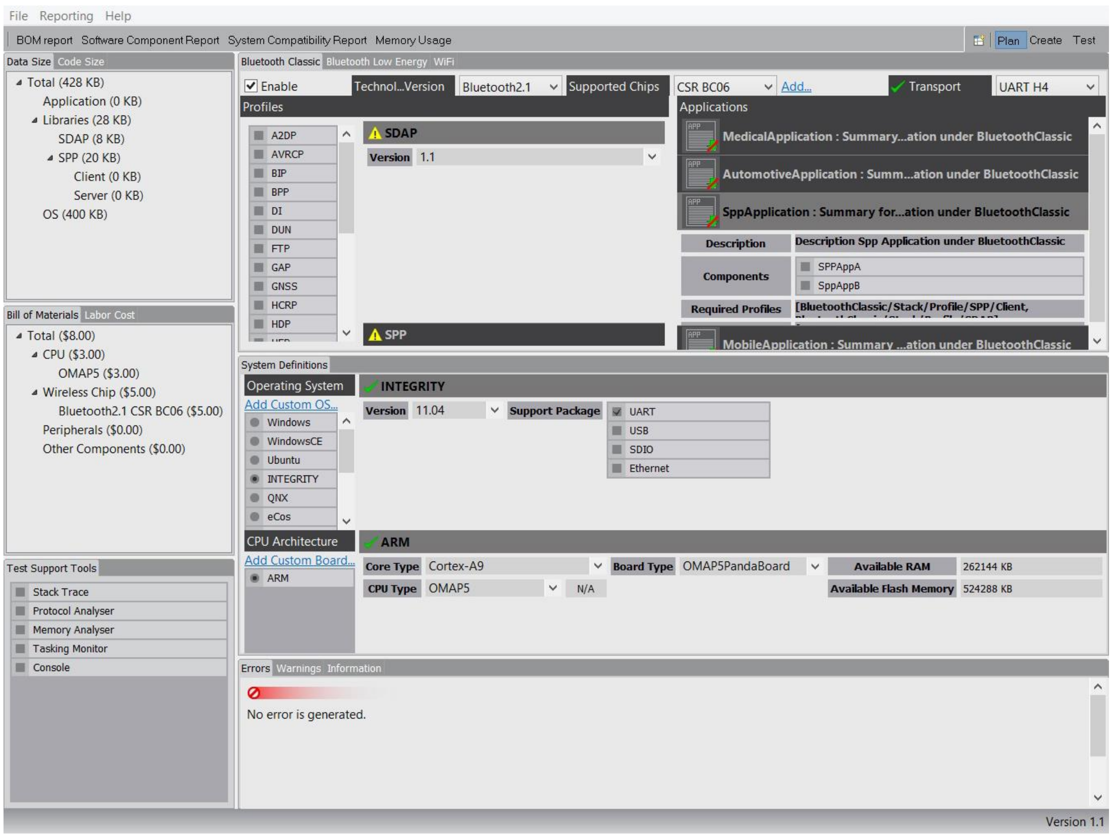
While multitasking has made the development of embedded wireless devices a little easier, RF technology is still difficult to plan, develop, test and debug. All the complications deflect the engineer's time and resource away from the thing that differentiates their end product from the competition – the application. Moreover, the low-level complexity and resulting design challenges can delay the product's market introduction.

One way to make things easier is to source the major software components from a proprietary software vendor rather than developing them from scratch. But even when taking advantage of this strategy, the options seem endless, making the correct selections challenging and merging the elements into a working application time consuming.

## *Intelligent Design Environment and framework ease wireless technology implementation*

Clarinox has developed a design environment software product called "Jannal". This could be thought of as a software interface, portal, design tool software, or even a Graphical User Interface (GUI); however the feature set, including decision support and facilitation of testing of multiple possibilities within a shortened time frame, lends these traditional descriptions to fall short. Jannal has been specifically developed to assist engineers in the design and development of complex embedded wireless devices and is thus more accurately described as Intelligent Design Environment software.

Jannal leads the engineer via an intuitive Eclipse-based graphical user interface (GUI) through the design phase tasks of plan, build and test, significantly reducing the efforts required to add wireless connectivity to a device. The engineer can choose either Bluetooth (including the new low energy variant) or Wi-Fi (see *figure 1*).


File Reporting Help
BOM report Software Component Report System Compatibility Report Memory Usage
Plan Create Test
Data Size Code Size
Bluetooth Classic Bluetooth Low Energy WiFi
Profiles
Applications
Bill of Materials Labor Cost
System Definitions
Test Support Tools
Add...
Add Custom OS...
Add Custom Board...
Errors Warnings Information
Version 1.1

Data Size
- Total (428 KB)
  - Application (0 KB)
  - Libraries (28 KB)
    - SDAP (8 KB)
    - SPP (20 KB)
      - Client (0 KB)
      - Server (0 KB)
  - OS (400 KB)

Bill of Materials
- Total ($8.00)
  - CPU ($3.00)
    - OMAP5 ($3.00)
  - Wireless Chip ($5.00)
    - Bluetooth2.1 CSR BC06 ($5.00)
  - Peripherals ($0.00)
  - Other Components ($0.00)

Test Support Tools: Stack Trace, Protocol Analyser, Memory Analyser, Tasking Monitor, Console

[✓] Enable | Technol...Version: Bluetooth2.1 | Supported Chips: CSR BC06 | ✓ Transport: UART H4

Profiles: A2DP, AVRCP, BIP, BPP, DI, DUN, FTP, GAP, GNSS, HCRP, HDP

SDAP — Version: 1.1

SPP

Applications:
- MedicalApplication : Summary...ation under BluetoothClassic
- AutomotiveApplication : Summ...ation under BluetoothClassic
- SppApplication : Summary for...ation under BluetoothClassic
- MobileApplication : Summary ...ation under BluetoothClassic

| | |
|---|---|
| Description | Description Spp Application under BluetoothClassic |
| Components | SPPAppA<br>SppAppB |
| Required Profiles | [BluetoothClassic/Stack/Profile/SPP/Client, |

Operating System: Windows, WindowsCE, Ubuntu, [✓ INTEGRITY], QNX, eCos

✓ INTEGRITY — Version: 11.04 | Support Package: [✓ UART], USB, SDIO, Ethernet

CPU Architecture: [✓ ARM]

✓ ARM — Core Type: Cortex-A9 | Board Type: OMAP5PandaBoard | Available RAM: 262144 KB | CPU Type: OMAP5 | N/A | Available Flash Memory: 524288 KB

No error is generated.

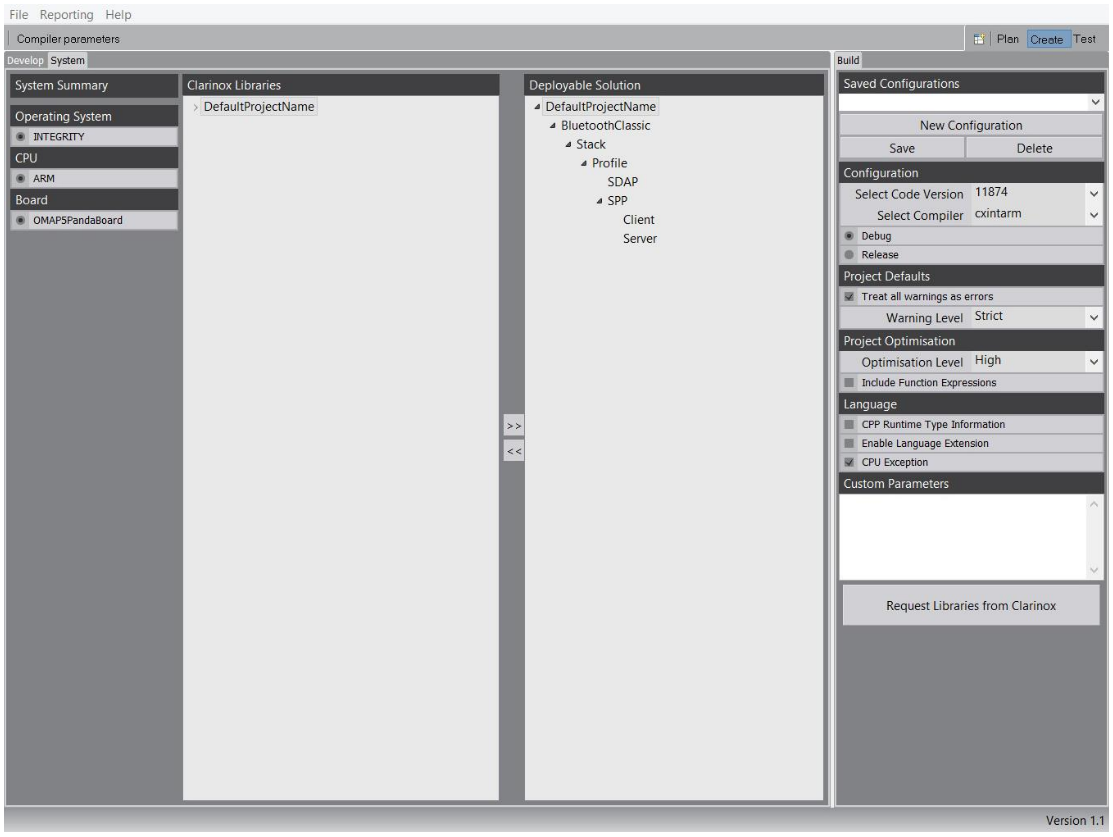
**Figure 1: Jannal guides the engineer through the planning phase**

A key advantage of the Jannal Intelligent Design Environment is the ease with which the engineer can try out alternative strategies in order to converge on the optimum design for his or her product. In addition to selecting the wireless protocol, the designer can experiment with standard profiles. It is possible to test the protocol with a variety of real time OSs (RTOS) and microprocessors (CPU) in order to explore various "what if?" scenarios to check if a selected design is robust and reliable under all envisaged operating circumstances. Without the flexibility of an Intelligent Design Environment, experimentation with such a wide range of variables would be impractical.

Once technology choices are made the engineer can proceed by choosing the compile/link parameters with the Create phase of the Intelligent Design Environment (see *figure 2*). Multiple combinations can be made and saved for later re-use. For example, a configuration can be built for debugging and later a separate configuration can be optimised for release. Automated compilation occurs upon request using Clarinox's server. Once downloaded, library code can be merged with application code and readied for testing.

The Jannal Intelligent Design Environment accesses debugging tools such as protocol analysers, memory statistics and memory leakage analysis. Operations such as internal event/message handling, task switching and thread switching can be analysed (see *figure 3*).

The Intelligent Design Environment can be used in different ways: One option is to incorporate existing Intellectual Property (IP) wireless protocol software elements and a standard sample application. In this case, the engineer doesn't need to learn the details of the underlying architecture. All that's required is an understanding of the application programming interface (API) set to be able to use the wireless protocol stack and the debugging tools.


File Reporting Help

Compiler parameters

Develop System

System Summary
- Operating System: INTEGRITY
- CPU: ARM
- Board: OMAP5PandaBoard

Clarinox Libraries
- DefaultProjectName

Deployable Solution
```
DefaultProjectName
  BluetoothClassic
    Stack
      Profile
        SDAP
        SPP
          Client
          Server
```

Build

Saved Configurations

Configuration
- Select Code Version: 11874
- Select Compiler: cxintarm
- Debug [✓]
- Release

Project Defaults
- Treat all warnings as errors [✓]
- Warning Level: Strict

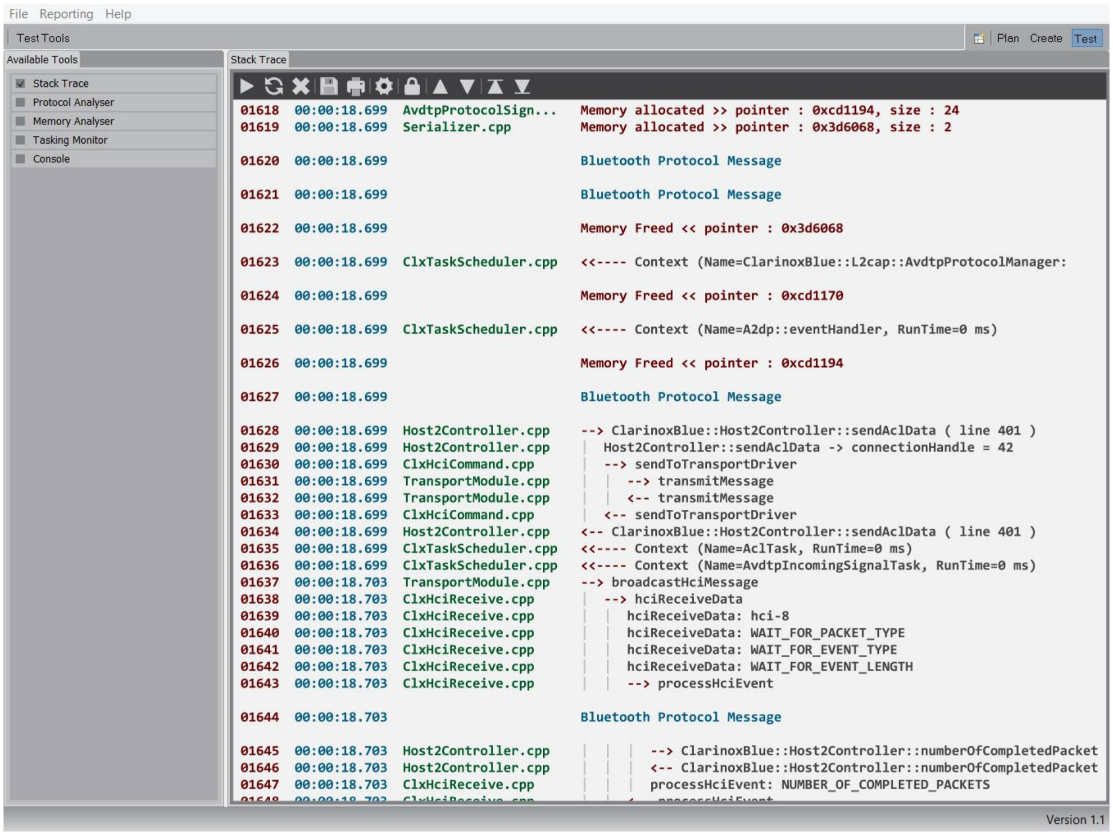
Project Optimisation
- Optimisation Level: High
- Include Function Expressions

Language
- CPP Runtime Type Information
- Enable Language Extension
- CPU Exception [✓]

Custom Parameters

Version 1.1

**Figure 2: Fine tune the selection and try alternative compilation options.**


File Reporting Help

Test Tools

Available Tools
- Stack Trace [✓]
- Protocol Analyser
- Memory Analyser
- Tasking Monitor
- Console

Stack Trace

```
01618  00:00:18.699  AvdtpProtocolSign...   Memory allocated >> pointer : 0xcd1194, size : 24
01619  00:00:18.699  Serializer.cpp         Memory allocated >> pointer : 0x3d6068, size : 2

01620  00:00:18.699                         Bluetooth Protocol Message

01621  00:00:18.699                         Bluetooth Protocol Message

01622  00:00:18.699                         Memory Freed << pointer : 0x3d6068

01623  00:00:18.699  ClxTaskScheduler.cpp   <<---- Context (Name=ClarinoxBlue::L2cap::AvdtpProtocolManager:

01624  00:00:18.699                         Memory Freed << pointer : 0xcd1170

01625  00:00:18.699  ClxTaskScheduler.cpp   <<---- Context (Name=A2dp::eventHandler, RunTime=0 ms)

01626  00:00:18.699                         Memory Freed << pointer : 0xcd1194

01627  00:00:18.699                         Bluetooth Protocol Message

01628  00:00:18.699  Host2Controller.cpp    --> ClarinoxBlue::Host2Controller::sendAclData ( line 401 )
01629  00:00:18.699  Host2Controller.cpp    |  Host2Controller::sendAclData -> connectionHandle = 42
01630  00:00:18.699  ClxHciCommand.cpp      |  --> sendToTransportDriver
01631  00:00:18.699  TransportModule.cpp    |  |  --> transmitMessage
01632  00:00:18.699  TransportModule.cpp    |  |  <-- transmitMessage
01633  00:00:18.699  ClxHciCommand.cpp      |  <-- sendToTransportDriver
01634  00:00:18.699  Host2Controller.cpp    <-- ClarinoxBlue::Host2Controller::sendAclData ( line 401 )
01635  00:00:18.699  ClxTaskScheduler.cpp   <<---- Context (Name=AclTask, RunTime=0 ms)
01636  00:00:18.699  ClxTaskScheduler.cpp   <<---- Context (Name=AvdtpIncomingSignalTask, RunTime=0 ms)
01637  00:00:18.703  TransportModule.cpp    --> broadcastHciMessage
01638  00:00:18.703  ClxHciReceive.cpp      |  --> hciReceiveData
01639  00:00:18.703  ClxHciReceive.cpp      |  |  hciReceiveData: hci-8
01640  00:00:18.703  ClxHciReceive.cpp      |  |  hciReceiveData: WAIT_FOR_PACKET_TYPE
01641  00:00:18.703  ClxHciReceive.cpp      |  |  hciReceiveData: WAIT_FOR_EVENT_TYPE
01642  00:00:18.703  ClxHciReceive.cpp      |  |  hciReceiveData: WAIT_FOR_EVENT_LENGTH
01643  00:00:18.703  ClxHciReceive.cpp      |  |  --> processHciEvent

01644  00:00:18.703                         Bluetooth Protocol Message

01645  00:00:18.703  Host2Controller.cpp    |  |  |  --> ClarinoxBlue::Host2Controller::numberOfCompletedPacket
01646  00:00:18.703  Host2Controller.cpp    |  |  |  <-- ClarinoxBlue::Host2Controller::numberOfCompletedPacket
01647  00:00:18.703  ClxHciReceive.cpp      |  |  |  processHciEvent: NUMBER_OF_COMPLETED_PACKETS
```

Version 1.1

**Figure 3: Test and debugging**

Another way of using the Intelligent Design Environment is to access the power of the proven framework, ClarinoxSoftFrame, which backs up Jannal as a modern, scalable architecture to guide the designer in the building of their application. The engineer will be required to develop a more detailed understanding of the protocol software and framework libraries, but by doing so it's possible for him or her to undertake a more ambitious approach when adding new ways of using wireless protocols.

ClarinoxSoftFrame framework "encapsulates" the protocol stack software and abstracts from underlying software elements such as the RTOS, allowing the engineer to develop the embedded device in relatively simple steps via its own API. (See Appendix B "*The advantages of a framework*".)

## An evolving discipline

Today's embedded system developers must deal with more complexity than even just a few years ago - and yet OEMs are under huge pressure to bring debugged and tested products to market in much reduced timeframes. Adding functionality demanded by consumers, such as a wireless communication protocol, makes the challenge just that little harder still.

That's where the advantages of an Intelligent Design Environment come to the fore. Jannal's underlying framework offers functionality that allows the developer to benefit from the relative simplicity of co-operative multitasking, while taking advantage of the flexibility of pre-emptive multitasking in challenging applications. This flexibility is particularly important for applications incorporating complex wireless communication protocols.

Jannal Intelligent Design Environment and ClarinoxSoftFrame framework allow the engineer to work on the wireless application layer, without having to worry about lower level issues such as dealing with the RTOS's synchronisation objects. It provides software tools, such as protocol analysis and memory management, to facilitate communication visibility and prevent issues such as memory leaks.

The Intelligent Design Environment, a new generation of software, allows experienced engineers with limited RF knowledge to develop embedded wireless devices without the need to learn the entire underlying system architecture. The software helps to manage the inherent complexity within the development process and frees up time for the engineer to experiment with various solutions for the end application.

In short, Jannal relieves the developer from much of the distraction that takes the focus away from the end application. More time to consider the end application positively encourages innovation - leading to winning products.

## *APPENDIX A:*

## *Pre-emptive scheduling vs. co-operative scheduling*

Multitasking applications are divided into “tasks” (or “processes”) that run simultaneously. If a single processor is used, control of the device must constantly rotate between the tasks that are ready to run. Scheduling which task has control of the processor at what time is a vital aspect of multitasking and using a scheduling algorithm unsuited to an application may significantly reduce the efficiency of that application in many ways.

All scheduling algorithms fall under one of two categories: Pre-emptive scheduling and co-operative scheduling.

During pre-emptive scheduling, the operating system (OS) forces a running task to relinquish control of the processor then schedules the next task to take control. An example of a pre-emptive scheduling algorithm is “Round Robin”. This algorithm uses time-slicing, in which each task has control of the OS for a predefined amount of time. After a task’s time-slice has expired, the OS takes back control and schedules the next task (“pre-empting”) in a defined order.

However, in co-operative scheduling, a running task continues to control the processor until it tells the OS, via a function call, that it can relinquish control of the processor. At this point, the OS takes control of the processor and schedules the next task to run. For example, “First-Come-First-Serve” (FCFS) is a co-operative scheduling algorithm in which the earliest task that is ready to run is given control of the processor and is run to completion. Once the process is terminated, the OS schedules the next task that is ready to run.

Tasks in a pre-emptive multitasking environment may be switched out at any time. This allows minimal response times for the systems, as higher priority tasks can pre-empt lower priority task execution. An unfortunate side effect is the uncertainty that arises from the developer not knowing when each task will be switched in or out by the OS. The developer must therefore ensure that the task, and every other task in the application, will continue to run smoothly (i.e. not dominate resources, be susceptible to memory corruption or result in deadlock) if it relinquishes control of the processor at any time. Such precautions to avoid these problems, such as synchronisation, may be difficult to implement due to the unpredictability of task switching, and bugs that subsequently arise may be challenging to track and fix, especially for very large applications.

As tasks in co-operative scheduling need function calls in order to terminate control of the processor, the developer knows at which point the task will be switched. Task switching is predictable, therefore debugging is much easier and the common problems such as deadlock and those involved in resource acquisition are easier to avoid than in a pre-emptive multitasking environment. However, response times may be longer than in a pre-emptive multitasking environment as the task that handles the input must wait for the executing task to relinquish control of the processor.

ClarinoxSoftFrame (see Appendix B) Co-operative Multitasking infrastructure provides multitasking abilities to the developer. Given that ClarinoxSoftFrame achieves co-operative multitasking under the use of a single thread, context switching by the processor need not occur, speeding up the system. The requirement to use synchronisation objects such as semaphores within a single thread is also eliminated. In addition to the ClarinoxSoftFrame Co-operative Multitasking infrastructure, a multithreading infrastructure is also available.

Multiple instances of both infrastructures may be utilised depending on the requirements of the system. ClarinoxSoftFrame gives the developer control over the scheduling by allowing him or her to decide which tasks get more execution time by setting task priorities. Together with the usual benefits of a co-operative scheduling environment, such as ease of debugging and ease of achieving task coordination, Clarinox SoftFrame simplifies the low level details of a project and allows the developer to concentrate on the end application.

## *APPENDIX B:*

## *The advantages of a framework*

Frameworks may be used to provide general functions to manage the complexities of system operation and “hide” the low-level functions such as some tasks of an operating system (OS) and those of communication protocols.

In the case of ClarinoxSoftFrame, the framework offers standard libraries and memory management functionality, wireless- and wired-communications protocols, input/output interfaces and a common interface to hardware such as a processor or software like an OS. This framework supplies elements to build a multitasking environment using a co-operative scheduling architecture (see Appendix A).

The product offers debug tools, which are presented to the user via a PC-based application, and are simple and clear to follow and use, reducing overall debug times. (For example, Clarinox Debugger is a PC-based debugging application that interfaces with ClarinoxSoftFrame and provides a dynamic link library (DLL)-based plug-in interface. This allows developers to easily add their own debugging functionality by defining their own plug-ins. The plug-in can pass specific messages sent by the debug target. The debug target can be connected to the debugger over various communication interfaces.)

A degree of standardisation adds the flexibility for an engineer to pick the best architecture for a particular circumstance. Moreover, as the interfaces to hardware are handled within the middleware, the developer does not need to change the application code when a new interface is employed. Consequently, the application can be easily updated to suit a new hardware platform.

By providing “pre-built” functions the developer doesn’t need as deep an understanding of system elements as is otherwise the case and is freer to focus on the end application.