How Embedded Applications using an RTOS can stay within On-chip Memory Limits

Robert Davis
Realogy
rdavis@realogy.com

Nick Merriam
Realogy
nmerriam@realogy.com

Nigel Tracey
University of York
njt@cs.york.ac.uk

www.realogy.com

Abstract

The major requirement from manufacturers designing embedded systems for high-volume products is to keep production costs as low as possible. The most cost-effective solution is to use small single chip microcontrollers / DSPs. Developers of software for resource constrained systems are increasingly in a difficult position: they have to write complex programs that must fit into very low cost microcontrollers and yet end-product quality must not suffer. This leads to a desire to use a real-time operating system (RTOS), yet remain within on-chip memory limits (typically 512bytes to 4K RAM). This is not possible with many conventional RTOS, which allocate a separate stack for each task. We ask the question "why don't they use a single stack?" and explain the key elements of an execution model required for single stack operation. The novel concept of non-preemption groups is introduced and shown to enable significant reductions in stack size. An effective priority allocation algorithm is essential to realize the advantages of non-preemption groups. The results of applying such an algorithm show that the number of preemption levels required to maintain schedulability is strongly correlated with the spread of deadlines and for typical systems is relatively small. We conclude that a priority allocation algorithm combined with the non-preemption group mechanism reduces stack requirements by a significant factor. This enables efficient operation using an RTOS, whilst remaining within on-chip memory limits. Stack requirements are similar to those for a cyclic executive.

INTRODUCTION

The major requirement from manufacturers designing embedded systems for high volume products is to keep production costs as low as possible. The most cost effective solution is to use small single chip microcontrollers or DSPs with the program executing out of on-chip ROM and using only the very limited amounts of on-chip RAM. This is borne out by the market for microcontrollers: in 1998, 2.5 billion 8-bit microcontrollers were sold, compared to only 63.7 Million 32-bit microcontrollers. The average selling price of an 8-bit device was \$2.23 compared to \$8.97 for faster 32-bit devices.

Adding external RAM leads to significant increases in cost. Even internal RAM is transistor hungry and expensive (8-16 times as costly as ROM). Typically, a number of microcontroller variants are produced with only those on-chip peripherals specifically required and with varying amounts of on-chip RAM. For example 256, 512, 768 or 1024 bytes of RAM on an 8-bit device (or up to a few Kbytes on an expensive 32-bit device).

The functionality and complexity of embedded software is constantly increasing in response to "more for less" demands by end customers. The challenge for manufacturers is to squeeze new functionality into the lowest cost hardware possible, without cutting corners on reliability and hence product quality.

The conventional way to build embedded software in resource constrained systems is to write a monolithic program where the logical software functions are broken into small pieces. These pieces are then run in a 'round robin' sequence. Building the sequence and writing the code for the functions turns out to be difficult to do when the number of functions becomes large and they become complex. It also starts to be difficult to make changes to the system, so incorporating new enhancements is time consuming and risky. As well as being difficult to write, the software ends up using the available processor time inefficiently, requiring a faster and more expensive microcontroller. [1].

Developers of software for resource constrained systems are increasingly in a difficult position: they have to write complex programs without impacting end-product quality. This leads to a need to use a real-time operating system (RTOS), yet the application must fit into very low cost microcontrollers.

STACK USAGE

Conventional RTOS approach

Real-time operating systems deal effectively with the complexity problems inherent in using 'superloop' or cyclic executive solutions. They do this by supporting multi-tasking and fixed priority preemptive scheduling. The advantages of using an RTOS are well known: there is no need to break functionality up into fragments to fit it within the major and minor cycles, functionality can be executed at the appropriate rate, and sporadic events such as infrequent but short deadline interrupts can be dealt with efficiently. Applications using an RTOS can make significantly more effective use of processor time than 'superloop' systems. Simple cyclic systems can waste as much as 20% of the CPU through over-sampling and provision for sporadic activities [1].

In a typical multi-tasking operating system, a number of tasks compete for use of the processor based on their priorities. When a low priority task is executing and a higher priority task becomes ready, the RTOS preempts execution of the low priority task, saves its execution context, restores the context for the high priority task which then executes. Once the high priority task has finished, the low priority task will be resumed.

Unfortunately, most RTOS allocate a separate stack to each task. This stack is used by the task as it executes to store local variables, return addresses etc. It may also be used when switching away from the task to save its context (current register values). Finally, when an interrupt is serviced, the interrupt handler may also execute on the task's stack. This separate allocation of a stack per task uses a large amount of RAM as shown by the example below: (Figures for an 8-bit micro with 512 bytes of on-chip RAM, context save of 15 bytes).

Task	**Priority**	**Stack** (bytes)
A	1	40
B	2	30
C	3	35
D	8	20
E	5	80
F	6	70
G	7	60
H	4	35
Interrupt	-	20

Table 1.

Total stack usage = Sum of all task stack usage + (N * context) + (N * interrupt stack usage).
= 650 bytes

In this example, the total amount of RAM needed for the task stacks is greater than the total on-chip RAM!

Note that an improvement can be made via the use of a separate stack for interrupts, however this typically requires hardware support (e.g. ARM7 devices) if it is to avoid placing any of the processor context on the task's stack.

Why use a stack per task?

The execution model used in conventional real-time operating systems can be traced back to operating systems designed for multi-processing in the 1960's. Each task loops indefinitely, waiting for some event (for example the expiry of its period).

```
void my_task()
{
   while(1) {
      /* do something */
      delay (500ms);
   }
}
```

This execution model means that the RTOS cannot use a single stack. If a single stack was used, then the following situation could occur: a low priority task L executes and its current state is placed on the stack. L is then preempted by a high priority task H, H uses the stack below the memory used by L (assuming that the stack grows downwards). Then when H voluntarily suspends via the delay() call, task L resumes. Either the stack pointer is incorrect for the resumption of L, or L overwrites the area of the stack currently in use by H.

As well as the basic execution model, the poor synchronization mechanisms such as semaphores or priority inheritance used by many RTOS also prevent the use of a single stack. For example, when priority inheritance is used to obtain mutual exclusion, the following situation can occur. Task L runs, obtains a resource and is then preempted by H. When H attempts to obtain the resource, it is suspended and the priority of L increased so that it can complete execution of its critical section. This however leads to the problem noted above: either the stack pointer is incorrect for the resumption of L, or L overwrites the area of the stack currently in use by H.

Requirements for Single Stack operation

Execution of multiple tasks on a single stack requires an execution model where each task executes to completion (possibly being preempted by higher priority tasks) before returning to any lower priority task. Periodic behavior is obtained by re-activating the task.

With single shot execution, once a low priority task L is preempted by a higher priority task H, H will execute to completion before L is resumed. This means that task H uses an area of the stack strictly below that currently in use by L and both can execute on the same stack.

A more effective mechanism for enforcing mutual exclusion is also required for single stack operation: this is the Stack Resource Policy [5], an enhancement of the Priority Ceiling Protocol [2]. Using this 'immediate' variant of the Priority Ceiling Protocol, each resource is allocated a priority, which is the highest priority of any task that is permitted to lock that resource. At run-time, when a low priority task L locks a resource shared with a high priority task H, its priority is immediately raised to the ceiling priority of the resource (at least as high as the priority of H). Hence task H cannot even start to execute until L has released all resources shared with H or higher priority tasks. Once task H starts to execute, it is therefore guaranteed to obtain all the resources it needs. This means that under the Immediate Priority Ceiling Protocol, resource locks are **not** blocking calls! So even in the presence of resource locks, tasks execute to completion, with strictly nested preemption by higher priority tasks. Operation is therefore possible on a single stack, and is also guaranteed to be deadlock free [5].

Single stack

Single stack operation alone, is not enough to make a significant reduction in the amount of stack used. This is illustrated by the set of tasks in table 1. With a single stack and a unique priority for each task:

Total stack usage = Sum of all task stack usage + (N * context) + interrupt stack usage.
= 510 bytes

There is an advantage in total stack usage over the multi-stack approach. Interrupt handler stack usage does not require space on multiple stacks, only on one. The interruption of any task results in the interrupt handler executing on the same stack, with no requirement for hardware support to switch to a special interrupt stack.

Non-Preemption Groups

With all tasks (and interrupt handlers) executing on a single stack, there is scope to reduce the amount of RAM required for the stack by a large factor. This can be achieved by limiting which tasks can preempt each other.

Control of preemption can be achieved with no additional run-time overheads via the use of Non-Preemption Groups. A Non-Preemption Group is simply a collection of tasks that are not permitted to preempt each other.

To support Non-Preemption Groups, each task has two static priorities:

1. Base priority
2. Dispatch priority

A task's base priority is the priority at which a ready task (which has not yet started to execute) competes for the processor.

The dispatch priority of a task A is the highest base priority of any task that shares a Non-preemption Group with task A. At run-time, when a task first starts to execute, the current active priority is set to its dispatch priority. This means that the task can only be preempted by tasks with a higher base priority than its dispatch priority. This means it cannot be preempted by any task with which it 'shares' a Non-Preemption Group.

In fact, Non-Preemption Groups are just like the Immediate Priority Ceiling Protocol. When a task first starts to execute, it is as if it had locked a resource shared with all other tasks in the same Non-Preemption Group. Therefore, it is guaranteed not to be preempted by any of them. It should be noted that the base and dispatch priority mechanism is more efficient than using resource locks to avoid preemption. It also avoids providing a window of opportunity for preemption to take place just prior to the task starting execution - leading to more nested contexts.

Using Non-Preemption Groups, the total stack usage can be significantly reduced:

Task	**Base priority**	**Dispatch priority**	**Stack** (bytes)
A	1	4	40
B	2	4	30
C	3	4	35
D	8	8	20
E	5	7	80
F	6	7	70
G	7	7	60
H	4	4	35
Interrupt	-		20

Table 2.

In the above example, there are two Non-Preemption Groups, consisting of tasks A ,B,C and H and tasks E,F and G respectively. This means that the worst case stack depth comprises the maximum stack usage of any one task from each Non-Preemption Group, plus the stack usage of task D. This system in effect has three (task) preemption levels.

Total stack usage = Max for each preemption level + (Num preemption levels * context) + interrupt stack usage.

= 205 bytes

Resource locking

As mentioned previously, Non-preemption Groups are very similar to resource locks under the Immediate Priority Ceiling Protocol. Therefore it is unsurprising to learn that resource locking can also result in decreased stack usage.

Task	**Base priority**	**Dispatch priority**	**Stack** (bytes)	
			(no locks)	**(with locks)**
A	1	4	10	40
B	2	4	30	
C	3	4	15	35
D	8	8	20	
E	5	7	40	80
F	6	7	40	70
G	7	7	60	
H	4	4	15	35
Interrupt	-		20	

Table 3.

In the above example, the maximum stack usage of tasks A, C and H occur when they have a resource locked which is shared with task G. The final two columns in table 3, break the stack usage down into the maximum used without the resource locked and the maximum with it locked. Similarly, tasks E and F have their worst-case stack usage whilst locking a resource shared with task D.

In this case, calculating the total worst-case stack usage is much more complex, however such calculations are easily amenable to automated tool support.

In this case, the maximum stack usage occurs when task B is preempted by task G, which is in turn preempted by task D.

= 175 bytes

(Leaving 337 bytes of on-chip RAM available for global variables, communications buffers and additional application functionality).

Clearly using a single stack, in conjunction with Non-Preemption Groups, can lead to a significant reduction in the amount of RAM required for the stack. However, placing tasks into Non-Preemption Groups also has an effect on system schedulability. By reducing the amount of preemption, it is possible that the system will start to miss deadlines. Integrated priority allocation and schedulability analysis addresses this problem.

PRIORITY ALLOCATION

A description of various priority allocation schemes for fixed priority preemptive scheduling can be found in [3]. These include Rate Monotonic Priority Allocation and Deadline Monotonic Priority Allocation, which are optimal for systems that meet a set of very restrictive criteria. They are not however optimal for real-world systems where tasks have:

1. Deadlines on some operation prior to their completion.
2. Deadlines larger than their minimum inter-arrival time.
3. Offset release points.

An optimal algorithm developed by Neil Audsley addresses some of these constraints [4].

This algorithm formed the basis of our work to devise an algorithm that allocates base and dispatch priorities to form schedulable allocations that use only a small number of preemption levels.

The problem of allocating priorities to obtain a system which is both schedulable and uses the minimum number of preemption levels is considerably harder than finding a schedulable allocation of unique priorities. The search space size is $O(2^{N-1}N!)$ where N is the number of tasks.

The algorithm devised at Realogy allows users to provide a parameter p to control complexity. With p set to 1, the algorithm is $O(N^2)$. With p=1, the algorithm is not always optimal, but it is highly effective. Larger values of p increase effectiveness at the expense of increased complexity $\sim O(N^{p+1})$.

The difficulty in achieving optimal priority allocation is amply illustrated by the following example, which shows that not only do Non-Preemption Groups sometimes improve schedulability, but that any algorithm used to find Non-Preemption Groups cannot simply place tasks one at a time in a Non-Preemption Group.

Task	Jitter	WCET	Period	Deadline
A	20	45	100	110
B	20	40	100	110

Table 4.

Consider the example given in table 4. Using simple fixed priority preemptive scheduling based on unique priorities: If task A is given the lower priority, then its response time is 145 making it unschedulable. Similarly, if task B is given the lower priority, then it has a response time of 150 and is therefore unschedulable.

If the two tasks are placed in a Non-Preemption Group, then the worst case response time for each task is 105 and they are therefore **both** schedulable.

The example in table 5 gives a further illustration of the difficulty in achieving an optimal priority allocation and the advantages of using both base and dispatch priorities.

Task	WCET	Period	Deadline
A	2	13	13
B	3	16	16
C	10	1000	1000

Table 5.

In this case, the system is schedulable with unique priorities. Placing tasks one at a time in a Non-Preemption group, task C can be placed at the lowest priority. If task A is then placed in the Non-Preemption Group, with B at a higher priority, we find that A's response time is 15, and it is therefore unschedulable. Similarly, if task B is placed in the Non-Preemption Group with A at a higher priority, then B's response time is 17, and it too is unschedulable. However, placing tasks A, B and C in the same Non-Preemption Group (with base priorities in deadline monotonic order) results in a schedulable system, with response times of 12, 15 and 15 respectively.

It is interesting to note that this system is not schedulable if all the tasks are considered as having the same priority and executed in FIFO order. In general, FIFO scheduling within a priority band is not an effective scheduling method. It leads to large blocking times, and hence long response times, as each task may have to wait for all of the others in the band to execute first

RESULTS

How many preemption levels are needed?

Analysis of the number of preemption levels required for systems containing 16 to 32 tasks was carried out using Realogy's Time Compiler tool. The Time Compiler provided both schedulability analysis and automatic priority allocation.

To provide the test data, 10000 systems were randomly generated. Deadlines for the tasks in each system where chosen from an exponential distribution with a spread of 5 orders of magnitude (representing deadlines from 10us to 1 second). For the purposes of this research, all tasks had their period set equal to their deadline. The execution time proportion for each task was chosen from a uniform distribution. The tasks execution time was then set to its normalized proportion of the total, multiplied by its deadline and the desired utilization. Systems that were found to be unschedulable with any priority allocation were discarded.

Figure 1 shows the number of preemption levels required and how this varies with system utilization, from 70% utilization to 95% utilization.

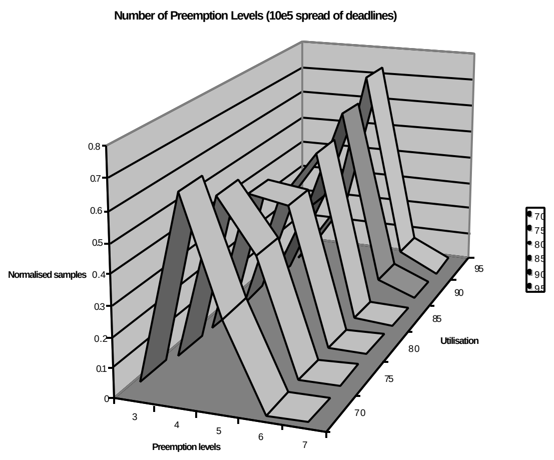

Figure 1. Number of preemption levels required at varying processor utilizations.

Figure 1 clearly shows that although there are up to 32 tasks in each system, with a 10^5 spread of deadlines, the maximum number of preemption levels required was 7. No systems were schedulable with just 1 or 2 preemption levels. (So using fixed priority non-preemptive scheduling none of these systems would be schedulable). Further, the average number of preemption levels needed changed from around 4 at 70-75% utilization to around 5 at 95-100% utilization.

Correlation with Number of tasks

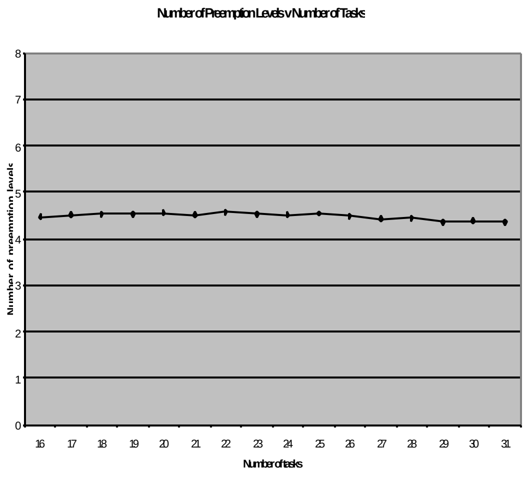

Figure 2. Number of preemption levels required for various numbers of tasks.

Figure 2 shows the average number of preemption levels required for various numbers of tasks, using the same data set as figure 1. The line is effectively flat, showing that the number of preemption levels is independent of the number of tasks, over the range 16 to 32.

Correlation with deadline spread

The analysis was then repeated. This time, the utilization of the systems was fixed at 80%, and the spread of deadlines was varied from 2 to 7 orders of magnitude.

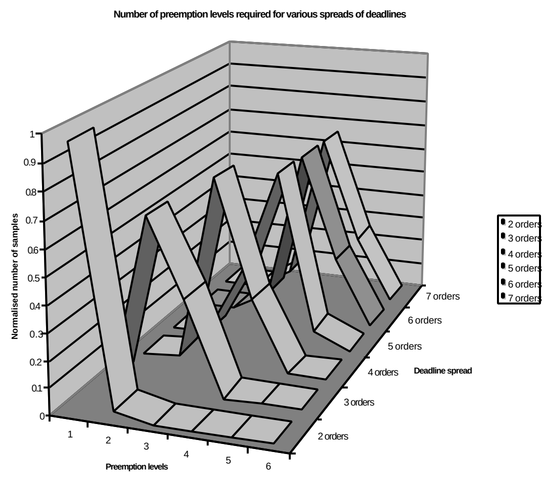

Figure 3 Number of preemption levels required for various spreads of deadlines.

Figure 3 shows the number of preemption levels required. It is interesting to note that very simple systems with only 2 orders of magnitude range of deadlines are schedulable with all the tasks in one Non-Preemption Group (corresponding to fixed priority non-preemptive scheduling). Whereas those with 3 or more orders of magnitude spread of deadlines require at least 2 or 3 preemption levels, with 4 or 5 preemption levels required for systems with a 10^5 to 10^7 spread of deadlines.

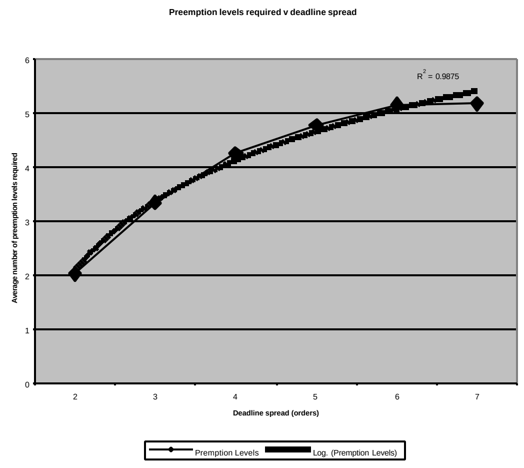

Figure 4 Correlation between number of preemption levels required and number of orders of magnitude covering all deadlines.

Figure 4 shows the strong correlation between the log spread of deadlines and the number of preemption levels required.

The number of preemption levels needed has a clear dependence on the spread of deadlines and is effectively independent of the number of tasks (assuming that there are enough tasks such that each order of magnitude spread of deadlines contains at least 2 or 3 tasks).

Reduction in stack size

For systems comprising 16 or more tasks, the Non-Preemption Group concept, combined with integrated priority allocation and schedulability analysis results in a reduction in the number of preemption levels required by a factor of 4 or more

The reductions in overall stack size can reasonably be expected to be of a similar scale to the reduction in the number of preemption levels. The actual reduction is dependent on the distribution of individual task stack usage and the amount of stack usage that is effectively overlapped due to resource locking effects.

SUMMARY

The major requirement from manufacturers designing embedded systems for high-volume products is to keep production costs as low as possible. This means staying within on-chip RAM and ROM limits.

Conventional RTOS approaches waste RAM by using a separate stack for each task. This problem can be addressed by using an RTOS that supports single stack operation. To be effective, this requires:

1. A single shot execution model.
2. Immediate Priority Ceiling Protocol for mutual exclusion.
3. Support for Non-Preemption Groups.
4. Automated schedulability analysis for real-world systems.
5. Integrated priority allocation algorithm.

Using Realogy's Time Compiler tools, the number of preemption levels and hence stack usage requirements of a typical application can be reduced to less than 25% of that required by an RTOS which requires a stack per task.

COMMON MISCONCEPTIONS

Academic

"Execution time is more important than memory"

Often academic work in the field of real-time systems focuses attention on execution time. The memory requirements of different scheduling methods rarely get a mention. However, in high volume embedded systems, on-chip RAM and ROM usage is the key factor. The difference in silicon area between an 8-bit and a 32-bit CPU core is relatively small when compared to adding kilobytes of extra RAM. This is exploited by ARM and other silicon vendors, with 32-bit CPU cores that can execute 16-bit instructions.

Only optimal algorithms are useful.

Most of the algorithms published in academic literature are optimal algorithms. Real-world problems that break the cozy assumptions needed for optimality also need effective solutions.

Fixed priority means a single fixed priority.

There is a wealth of potential in using 'fixed' priorities but changing the priority of tasks at key points e.g. when they start to execute. Research in this area may well move the state of the art in fixed priority preemptive scheduling theory forward.

Industrial

Cyclic executives are efficient

The actual code for a cyclic executive may be very small and fast. This does not however mean that the most effective use is being made of the CPU. Often large amounts (20%+) of CPU time are wasted through over-sampling and allowing for short deadline sporadic events. Compared to 1-3% overheads for an efficient RTOS [1].

RTOS have large overheads (execution time, RAM and ROM).

This is certainly true of many operating systems, which have their roots in mainstream computing. Multi-stack RTOS are inherently wasteful of RAM. With a carefully designed single stack RTOS, supported by integrated priority allocation and schedulability analysis tools, overheads can be reduced to a few percent of CPU time and a fraction of on-chip memory. For example, RTOS overheads of 1009 bytes of ROM and 79 bytes of RAM (135 bytes total stack usage) for a 10-task benchmark on a Motorola 68HC12.

Using priority allocation and Non-Preemption Groups, RAM usage can be as low as using a cyclic executive.

If I want to analyze a system for timing correctness, then I have to use a cyclic executive.

Using an analyzable RTOS and co-designed schedulability analysis tools, it is possible to perform full timing analysis on real-world embedded systems.

KEY CHALLENGES

Industry needs tool support:

1. Analyzable RTOS.
2. Schedulability Analysis tools (with support for real-world systems).
3. Priority allocation algorithms (don't care what the allocation is, but the system must fit on chip and meet its deadlines).
4. Worst case execution time analysis and measurement tools.

REFERENCES

[1] A. Hutcheon, From Cyclic Schedule to Preemption: Issues and Benefits, Miller Freeman Embedded Seminar, the Moller Centre, Cambridge 6th April 2000. Available at www.realogy.com

[2] L. Sha, R. Rajkumar, J. Lehoczky, Priority Inheritance Protocols: An approach to Real-Time Synchronization, IEEE Transactions on Computers 39(9) Sept. 1990 pp1175-1185.

[3] N.C. Audsley, A. Burns, R.I.Davis, K.W.Tindell, A.J.Wellings, Fixed Priority Scheduling: An Historical Perspective, Journal of Real-Time Systems, 8(2/3): 129-154, 1995

[4] N.C. Audsley, Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times, Technical Report YCS-164, Dept. of Computer Science, University of York, England. November 1991. Available at ftp://ftp.cs.york.ac.uk/pub/realtime/papers/YCS164.ps.Z

[5] T.P. Baker, A Stack Based Resource Allocation Policy for Real-time Processes" Proceedings 11th IEEE Real-Time Systems Symposium 1990 p191-200.

CONTACT DETAILS

Realogy,
Innovation Centre,
York Science Park,
York, England,
YO10 5DG

www.realogy.com