A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems

BRUCE BELSON, James Cook University, Australia
JASON HOLDSWORTH, James Cook University
WEI XIANG, James Cook University
BRONSON PHILIPPA, James Cook University

Many Internet of Things and embedded projects are event-driven, and therefore require asynchronous and concurrent programming. Current proposals for C++2020 suggest that coroutines will have native language support. It is timely to survey the current use of coroutines in embedded systems development. This paper investigates existing research which uses or describes coroutines on resource-constrained platforms. The existing research is analysed with regard to: software platform, hardware platform and capacity; use cases and intended benefits; and the application programming interface design used for coroutines. A systematic mapping study was performed, to select studies published between 2007 and 2018 which contained original research into the application of coroutines on resource-constrained platforms. An initial set of 566 candidate papers, collated from on-line databases, were reduced to only 35 after filters were applied, revealing the following taxonomy. The C & C++ programming languages were used by 22 studies out of 35. As regards hardware, 16 studies used 8- or 16-bit processors while 13 used 32-bit processors. The four most common use cases were concurrency (17 papers), network communication (15), sensor readings (9) and data flow (7). The leading intended benefits were code style and simplicity (12 papers), scheduling (9) and efficiency (8). A wide variety of techniques have been used to implement coroutines, including native macros, additional tool-chain steps, new language features and non-portable assembly language. We conclude that there is widespread demand for coroutines on resource-constrained devices. Our findings suggest that there is significant demand for a formalised, stable, well-supported implementation of coroutines in C++, designed with consideration of the special needs of resource-constrained devices, and further that such an implementation would bring benefits specific to such devices.

CCS Concepts: • **Software and its engineering → Coroutines**; *Compilers*; • **Computer systems organization →** *Embedded software*; Sensors and actuators;

Additional Key Words and Phrases: embedded, resource-constrained, asynchronous, direct style, scheduling

ACM Reference Format:
Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. 2019. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. *ACM Trans. Embedd. Comput. Syst.* 1, 1 (February 2019), 22 pages.

Authors' addresses: Bruce Belson, James Cook University, College of Science & Engineering, Cairns, Queensland, 4870, Australia, bruce.belson@my.jcu.edu.au; Jason Holdsworth, James Cook University, College of Business, Law & Governance, jason.holdsworth@jcu.edu.au; Wei Xiang, James Cook University, College of Science & Engineering, wei.xiang@jcu.edu.au; Bronson Philippa, James Cook University, College of Science & Engineering, bronson.philippa@jcu.edu.au.

1539-9087/2019/2-ART $15.00
https://doi.org/

1 INTRODUCTION

The Internet of Things (IoT) [Al-Fuqaha et al. 2015; Atzori et al. 2010; Gubbi et al. 2013] continues to grow both in the scale and variety of attached devices and in the number of developed applications [Manyika et al. 2015; van der Meulen 2017]. This growth draws attention to the software engineering of the resource-constrained embedded systems that are a frequent component of heterogeneous IoT applications~~,~~. Such attention is all the more urgently required because of new challenges with regard to security [Sicari et al. 2015], reliability [Gubbi et al. 2013] and privacy [Weber 2015].

Many IoT and embedded systems ~~software~~ have an event-driven architecture~~and~~; their software is consequently implemented in an asynchronous programming style, whereby multiple tasks wait on external events. Asynchronous code is challenging to write because application logic becomes split between the function initiating the request and the event handler that is invoked when the response is ready [Gay et al. 2003; Levis and Culler 2002; Meijer 2010]. This "split-phase" architecture becomes increasingly complex when the developer introduces more event sources (such as timeouts) with their own event handlers. There may be interaction between various split-phase events, which can add degrees of freedom to the various state models: consequently there is an increasing likelihood that the source code addressing a single event is split between separate locations, forcing the reader to jump between them. Application logic is obscured by the split-phase fragmentation, leading to a gap between the design of the system and its source code representation, making the code harder to understand and more difficult to maintain [Brodu et al. 2015; Edwards 2009; Kambona et al. 2013; Madsen et al. 2017].

A solution to the split-phase problem for desktop software has been language support for coroutines [Conway 1963; Knuth 1997; Marlin 1979] and promises [Brodu et al. 2015; Liskov and Shrira 1988; Madsen et al. 2017]. For example, in C#, JavaScript, and Python, developers can use an "await" keyword to wait on an external event. This means that asynchronous code can be written just as clearly as the equivalent code in a synchronous style that uses blocking code. However, resource-constrained embedded systems are overwhelmingly programmed in C or `C++` [AspenCore Global Media 2017; Skerrett 2017], which lack support for the "await" pattern.

The C++ standardisation committee is currently debating the inclusion of coroutines, and at least two competing designs have been proposed [ISO/IEC 2017; Romer et al. 2018]. The addition of coroutines to C++ would create an opportunity to simplify embedded systems code.

Existing research on coroutines in C++ may not have considered the needs of embedded systems and other extremely resource-constrained devices, because the initial implementations used compilers that do not target such platforms [Mittelette 2015]. Here, we specifically focus on small embedded systems that have insufficient memory to run Linux or another general purpose OS. If the C++ language adds the async/await and coroutine patterns, we believe it is important that the needs of resource-constrained platforms are also considered.

This article contains a systematic mapping of the use of coroutines in embedded systems, conducted by searching academic databases and manually inspecting every matching paper. It thus provides a complete perspective on academic research addressing the use of coroutines in embedded systems to inform the C++ standardization process by identifying how and why coroutines are used. The article uses the mapping to build a taxonomy of existing research with regard to platform, use cases and implementation.

The design of the study, details of the methodology used for each stage and the results of each stage are available in spreadsheet format. The remainder of this paper is organised as follows. Section 2 contains the background~~.~~, beginning with an introduction to the development environment for C/C++ programs on resource-constrained devices, to some of the problems commonly encountered by developers and to the types of solution currently applied to these problems~~.~~. It continues with

a discussion of the use of coroutines in C and C++~~; and a review of related work~~. Section 3 details the methodology of the mapping process used in this study, ~~and~~ some of the logic underpinning the methodological choices, and a review of related work. Section 4 explores the results and presents insights. Section 5 contains a discussion of results and an analysis of research gaps. Section 6 discusses future research possibilities and concludes the paper.

2 BACKGROUND

2.1 Async/Await pattern

Much of the program flow in IoT and embedded device programs is asynchronous, for example, requiring the software to wait on responses from a remote device or sensor. A naïve approach to implementing this flow results in complex arrangements, such as a finite state machine (FSM) and multiple fragments of code. This produces source code that is complex, fragile and inflexible.

Alternatively, there are two common patterns for a simpler and more robust design. The first, continuation-passing style (CPS), which is seen commonly in JavaScript, can lead to the "pyramid of doom" or "callback hell" phenomenon [Brodu et al. 2015; Edwards 2009; Kambona et al. 2013; Madsen et al. 2017] when multiple sequential operations are composed.

A more elegant approach is the async/await pattern [Bierman et al. 2012; Haller and Zaugg 2016; Okur et al. 2014; Syme et al. 2011], which is a popular device for transforming continuation-passing style code into direct programming style, with all the asynchronous steps of a sequence written in a single ordered sequence within a single block of code. The pattern has been used successfully in several languages, notably `C#` [Bierman et al. 2012; Okur et al. 2014] and JavaScript, as part of the ECMAScript 2018 proposal [ECMA 2017]; in C++, proposals are currently being considered for inclusion in the `C++ 2020` standard, using new keywords 'co_await', 'co_yield' and 'co_return' or alternative syntax [ISO/IEC 2017; Romer et al. 2018]. The async/await pattern allows the programmer to write a single continuous set of statements in a direct programming style, which will be performed in the correct order, even when they are run asynchronously as a set of separate events. Furthermore, the pattern avoids the explicit use of global variables.

2.2 Coroutines

Coroutines extend the concept of a function by adding suspend and resume operations [Conway 1963; Knuth 1997; Marlin 1979]. Coroutines can be used for a variety of purposes including (i) event handlers [Dunkels et al. 2006]; (ii) data-flow [Kugler et al. 2013]; (iii) cooperative multitasking [Susilo et al. 2009] as well as (iv) the async/await pattern [ISO/IEC 2017].

During suspension, the implementation stores the execution point of the coroutine, and usually (but not always) the state of local variables. For example, Protothreads [Dunkels et al. 2006] is a coroutine implementation for embedded systems where local variables are not preserved: instead all variables within the coroutine must be statically allocated. This strategy reduces the overhead of context switching and provides predictable memory usage but produces coroutines that are non-reentrant. Furthermore, code defects are more likely when the programmer is responsible for explicitly managing coroutine state. This study will examine both types of coroutine in the context of embedded systems.

Coroutine implementations may be further categorised into stackful or stackless types. A stackful coroutine has its own stack which is not shared with the caller, and hence local variables can be stored there during suspension. Conversely, a stackless coroutine pops its state off the stack during suspension (like a normal function return). ~~Neither model is considered universally appropriate~~

~~for the various C++ use cases [Goodspeed 2014; Riegel 2015].~~ For stackless coroutines, other mechanisms must be introduced in order to preserve state~~.~~, such as storing local variable in global storage.

Furthermore, a stackless coroutine ~~cannot~~ often can only be suspended from ~~a subroutine; the yield must take place in~~ within the coroutine itself and not from a subroutine (a function called from the coroutine). For example, C++ proposal N4680 is a stackless model that requires all yield or return statements to be contained within the body of the coroutine~~itself.~~ .

Neither model is considered universally appropriate for the various C++ use cases [Goodspeed 2014; Riegel 2 . Alternative techniques, such as stack slicing, have been used to preserve state in a stackless implementation and provide single threaded cooperative multitasking [Tismer 2000, 2018].

2.3 Previous coroutine implementations for constrained platforms

~~In 2000, Tatham [Tatham 2000]~~ Early implementations used macros in C to add coroutine-like features. For example, Duff's device takes advantage of the fall-through behaviour of C's case statement in the absence of a break statement [Duff 1988]. It is unusual in that a block such as do ... while, can be interleaved within the case statements of a switch statement. Tatham described a coroutine solution in C, which makes use of Duff's device ~~[Duff 1988]~~ to efficiently implement coroutines through macros, without the need to explicitly code a state machine~~, and thereby satisfies the producer-consumer pattern~~[Tatham 2000]. However, Tatham noted that "this trick violates every coding standard in the book" and Duff called the method a "revolting way to use switches to implement interrupt driven state machines". This technique was extended by Dunkels et al. for Protothreads [Dunkels et al. 2006], which provided conditional blocking operations on memory-constrained systems, without the need for multiple stacks, and formed the core of the widely used real-time operating system Contiki [Dunkels et al. 2004].

Protothreads (and any other solution based on Duff's device) can be considered to suffer from two serious defects. First, their use adds a serious constraint to C programs: switch statements cannot be used safely in programs that use Protothreads; they may cause errors that are not detected by the compiler but cause unpredictable behaviour at run-time. Second, they do not manage local variable state on behalf of the programmer: any variable within the coroutine whose state should be maintained between calls must be declared as static (global) [Dunkels 2005]. This has consequences for reentrancy, and for code quality. On the other hand, they are an extremely cheap solution in terms of coding effort, memory use and speed, and they are portable, because they use pure C. The original library is written in C; it has been ported to C++ [Paisley and Sventek 2006].

Listing 1 contains a fragment of code that used Protothreads to implement part of an asynchronous producer/consumer pattern. Listing 2 shows a similar code fragment, this time using C++ language features, including the co_await keyword of the current C++ standardisation proposal N4680. We observe several differences between the two: Listing 2 contains fewer lines of code than Listing 1; Listing 2 does not contain macros; Listing 1 requires that local variables be declared as 'static', but Listing 2 does not. While both code fragments present conceptual changes from the synchronous equivalent, we believe that the change in Listing 2 is more transparent and is more clearly presented.

Listing 1. Fragment of Protothreads code for asynchronous producer/consumer threads

```
static struct pt_sem full, empty;

static
PT_THREAD(consumer(struct pt *pt))
{
```

```
  static int consumed;
  PT_BEGIN(pt);
  for (consumed = 0; consumed < NUM_ITEMS;
       ++consumed) {
    PT_SEM_WAIT(pt, &empty);
    consume_item(get_from_buffer());
    PT_SEM_SIGNAL(pt, &full);
  }
  PT_END(pt);
}
```

Listing 2. C++ code fragment using co_await for asynchronous producer/consumer threads

```
task<> consumer(semaphore& sem) {
  auto producer = async_producer(sem, NUM_ITEMS);
  for co_await(auto consumed : producer) {
    consume_item(get_from_buffer());
  }
}
```

In 1992, Gupta et al. examined a coroutine-based concurrency model for resource-constrained platforms as part of a comparison between alternative models [Gupta et al. 1992]. In 2000, Engelschall summarised the various techniques based on setjmp & longjmp [Engelschall 2000]. FreeRTOS [Barry 2018] is an open source real-time operating system developed "around 2003" that contains a coroutine scheduler: local variable state is not maintained. In 2006, Rossetto and Rodriguez described a new concurrency model [Rossetto and Rodriguez 2006] implemented as an extension to TinyOS [Levis et al. 2005], using coroutines as the basis of the integration; the implementation is stackful and local variables' states are maintained.

Schimpf [Schimpf 2012] provides a modified version of Protothreads which supports a priority-based scheduler. Cohen et al. [Cohen et al. 2007] provide a coroutine-based scheduler for TinyOS [Levis et al. 2005] which is used to implement "RPC-like interfaces"; these support a direct programming style for communications code written in nesC [Gay et al. 2003]. Riedel et al.[Riedel et al. 2010] generate C code for multiple platforms, including a version that uses coroutines to provide concurrency. Susilo et al. [Susilo et al. 2009] use a coroutine-based scheduler to achieve "[r]eal time multitasking [...] without interrupts". Finally, Andersen et al. [Andersen et al. 2017] reject the use of C++ futures because the implementation model needs to handle a stream of events, rather than a single event, and is therefore non-deterministic in its use of memory, which is undesirable on a constrained platform: "One is forced therefore to trade off the reliability of promises [...] in order for them to work in the embedded space." Instead, the authors use callbacks for C++ event handling.

The scripting language Lua possesses a coroutine implementation [Moura et al. 2004] and has been successfully used on microcontrollers [Hempel 2008]. MicroPython [George 2014] is a Python 3 version which supports microcontrollers [George and Sokolovsky 2014] and includes support for generators and coroutines [The Python Software Foundation 2018; van Rossum and Eby 2005].

2.4 Programming Languages: C and C++

The majority (78%) of embedded systems are programmed in C or C++ according to the 2017 Embedded Markets Study [AspenCore Global Media 2017]. The C language is the most popular, but its usage is slowly declining over time in favour of C++ and other languages. Between 2015 and 2017 the proportion of embedded projects using C++ rose from 19% to 22%, while C use fell

from 66% to 56%. Coroutines are proposed for the C++ language, not C, so embedded programmers would need to use C++ to access these features. We believe that C++ usage will continue to increase, and therefore the design of C++ coroutines should consider the constraints of embedded software.

The switch from C to C++ need not be dramatic. C++ is close to being a superset of C [Stroustrup 1986]. With the right compiler support, it is possible to migrate an embedded code-base from C to C++ merely by changing a compiler flag. There are potential problems with the migration from C to C++, including the possibilities that the code produced may be larger, slower and less likely to contain blocks that are appropriate for placement in ROM than the C code [Goldthwaite 2006; Herity 2015]. A more subtle problem is that the code may be less amenable to worst-case execution time (WCET) analyses. Goldthwaite [Goldthwaite 2006] examined these problems, and identified three areas where difficulties might exist despite defensive programming, all of them with regard to timing analysis: (i) dynamic casts, (ii) dynamic memory allocation and (iii) exceptions.

The features that might persuade a development team to make the move to C++ have always included well-known front-end features such as namespaces, encapsulation, and inline functions, all of which offer benefits regarding code clarity but have no implementation cost in terms of code size or speed. Replacing split-phase functions with a direct programming style by using new, widely supported, language standards would appear to be a strong enticement for developers to migrate. It remains to be seen whether the feature can be provided for embedded systems without including two of the three behaviours which Goldthwaite [Goldthwaite 2006] identified as being problematical: dynamic memory allocation and exceptions.

3 SYSTEMATIC MAPPING STUDY

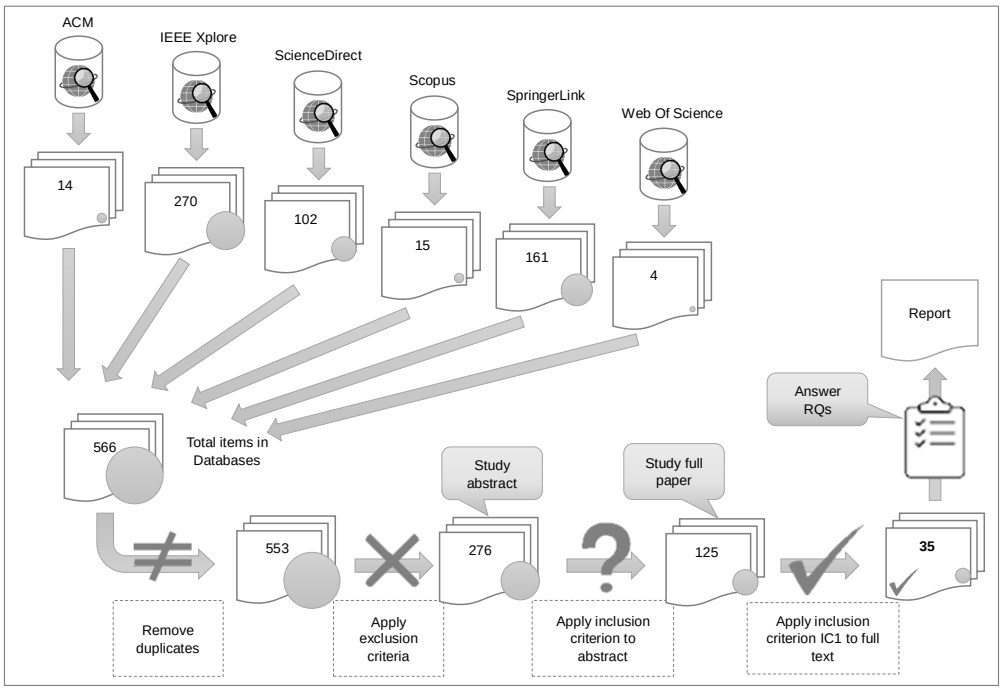


Fig. 1. Summary of the search and selection process

3.1 Overview

This systematic mapping study is informed by the guidelines of Kitchenham [Kitchenham 2004], Kitchenham and Charters [Kitchenham and Charters 2007] and Petersen et al. [Petersen et al. 2008]. The process is illustrated in Figure 1. The process searched six online databases, selected for relevance [Brereton et al. 2007] and availability, for papers containing a term from each of the lists in Table 1. We ensured completeness by iterative testing using snowballing [Kitchenham et al. 2011; Petersen et al. 2015] and by careful handling of database-specific behaviours regarding plurals, spellings and abbreviations.

Table 1. Search strings used for online databases

Part 1: Pattern	Part 2: Platform
coroutine OR "lightweight thread"	AND IoT OR "Internet of Things" OR "Cyber Physical Systems" OR RTOS OR "Real-time Operating Systems" OR "Embedded Systems" OR WSN OR "Wireless sensor networks" OR WSAN

3.2 Search procedure

The main inclusion criterion was that the paper should contain original research into the application of coroutines on resource-constrained platforms (IC1). This criterion excluded a large body of papers which applied coroutines only within the simulation of resource-constrained platforms, not on the platform itself.

The exclusion criteria were informed by previous studies [Kitchenham and Charters 2007; Petersen et al. 2008]. Papers were excluded if they lacked a scholarly identifier such as DOI or ISBN (EC1) or an abstract (EC2), were published before 2007 (EC3), were not written in English (EC4), were not available to the reviewers (EC5), were earlier versions of another paper (EC6), were not primary studies (EC7) or were not in any of the selected publication classes (journal articles, conference papers or book chapters) (EC8).

~~The search,~~ Two searches were conducted in October 2017 and in September 2018 across all databases~~, found~~ ; together, these searches resulted in 187 journal articles, 224 conference papers and 155 book chapters. This informed our decision to include all three publication classes within the search domain. The decision was made to include only studies published since 2007; this criterion excluded approximately 43% of the original search results.

Details of the search strings, inclusion and exclusion criteria, procedures and download scripts can be found in the supplementary materials.

3.3 Other systematic reviews and mapping studies

An initial tertiary study was executed, being a review of existing reviews and mapping studies in the area of interest, as suggested by Kitchenham and Charters' guidelines [Kitchenham and Charters 2007]. The work concluded that, at the time of writing, this study appears to be the first to systematically map the use of coroutines in resource-constrained systems, whether as embedded systems or as IoT component systems.

3.4 Research questions

A major motivation for the study was to prepare the ground for an acceptable implementation of the await/async and generator patterns on resource-constrained platforms, using coroutines.

The research questions therefore address what is known about hardware and software platforms, developer preferences, use cases, intended benefits, and application programming interfaces (APIs).

RQ1 investigated the software platform, including ~~programing language,~~ the programming language, the operating system and the implementation used for the relevant language feature. RQ2 looked at the hardware platform, including memory size and processor family. RQ3 and RQ4 assessed the use cases and intended benefits respectively of the coroutine usage. RQ5 assessed the programming interface.

The research questions are listed in full in ~~the supplementary materials~~Table 2. By examining the hardware and software characteristics of previous implementations we aimed to identify the salient characteristics of the environment within which a coroutine implementation must function; by investigating use cases and desired outcomes we would identify some of the necessary characteristics of a successful implementation; finally, by examining the programming interface we hoped to observe how researchers addressed some of the design trade-offs of the implementation.

Table 2. Research questions

Code		Research question
RQ1		What was the software platform?
	RQ1a	What was the programming language used?
	RQ1b	What method was used to implement coroutines?
	RQ1c	What was the operating system used (if any)?
RQ2		What was the hardware platform?
	RQ2a	What was the class of hardware platform?
	RQ2b	How much read-only or flash memory (ROM) was available?
	RQ2c	How much random-access memory (RAM) was available?
	RQ2d	What was the processor family?
	RQ2e	Was the processor 8-bit, 16-bit or 32-bit?
	RQ2f	What was the processor's instruction set?
RQ3		What were the use cases?
RQ4		What were the intended benefits of using coroutines in this context?
RQ5		What is the API of the coroutine?
	RQ5a	Does the paper discuss an implementation of coroutines?
	RQ5b	Is the control flow managed on behalf of the developer?
	RQ5c	Is the state of local variables automatically managed?
	RQ5d	Is the coroutine implementation stackless or stackful?
	RQ5e	How is the coroutine state allocated?

3.5 Threats to validity

Data extraction followed the principles laid down in Petersen et al. [Petersen et al. 2015] for repeatability.

The validity of the results of this study are exposed to multiple sources of threat, particularly with regard to (i) study selection, (ii) data extraction and (iii) classification.

During study selection, the search process was recorded in detail and the search strings were tested for repeatability and for consistency across databases. Snowballing describes the process of expanding the search results by recursively selecting papers cited by a selected paper or which cite a selected paper [Kitchenham et al. 2011; Petersen et al. 2015]. While the study did not utilise snowballing ~~[Kitchenham et al. 2011; Petersen et al. 2015]~~ during the final search process, it did

use ~~snowballing~~ it during the earlier stages of establishing search strings, and some searches were consequently amended. During the application of selection criteria, the reviewers ~~consulted~~ conferred whenever differences arose, and periodically discussed and reviewed the processes being used, using both contentious cases and randomly selected test papers to compare individual processes.

The guidelines of Petersen et al. [Petersen et al. 2008, 2015] were followed with regard to the data extraction process: a data collection form was constructed in Excel, and was used consistently to record the process, in order to improve repeatability and accuracy, and to reduce subjectivity.

To improve the consistency of classification a subset of papers was inspected by both reviewers, and the classifications were compared and discussed. This comparison was iterated until the rationale for classifications was fully established and any contentious cases had been resolved.

3.6 Data set

The initial search found 566 results; removal of duplicates left 553 documents. More than half of these failed the exclusion criteria, leaving 276 whose abstracts were studied.

Approximately 55% of the surviving studies immediately failed the inclusion criteria, leaving 125 to be studied in full. After applying the inclusion criteria on the basis of the entire text, about 72% failed, and 35 studies were retained [Alvira and Barton 2013; Andalam et al. 2014; Andersen et al. 2016, 2017; Bergel et al. 2011; Boers et al. 2010; Clark 2009; Cohen et al. 2007; Durmaz et al. 2017; Elsts et al. 2017; Evers et al. 2007; Fritzsche and Siemers 2010; Glistvain and Aboelaze 2010; Inam et al. 2011; Jääskeläinen et al. 2008; Jahier 2016; Kalebe et al. 2017; Karpinski and Cahill 2007; Khezri et al. 2008; Kugler et al. 2013; Kumar et al. 2007; Liu et al. 2011; Lohmann et al. 2012; Motika and von Hanxleden 2015; Niebert and Caralp 2014; Noman et al. 2017; Oldewurtel et al. 2009; Park et al. 2015; Riedel et al. 2010; Schimpf 2012; St-Amour and Feeley 2010; Strube et al. 2010; Susilo et al. 2009; von Hanxleden 2009; Yu et al. 2008]. Of these, 21 studies included a discussion of the implementation of coroutines. The lower half of Figure 1 illustrates the process.

The selected papers addressed the issue of coroutines despite the lack of mainstream language support. These researchers identified a need that was not addressed by common languages and showed the potential benefits of these features. Now that native asynchronous programming support is being added to the C++ language, it is likely that demand from embedded software developers will only increase.

4 RESULTS

4.1 Overview

The research identified 35 papers of relevance, of which 21 described coroutine implementations, developed in 7 different programming languages. Detailed lists of the results may be found in the supplementary materials.

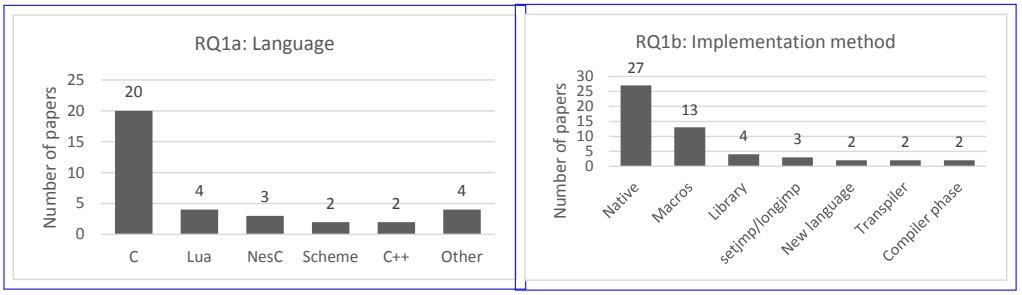
4.2 Programming language

~~RQ1a – Programming languages used~~ ~~RQ1b – Coroutine implementation~~ ~~Language outcomes~~

C was the predominant programming language, as shown in Figure 2a. 20 of the papers (57%) used C, and a further 5 papers (14%) used related languages (C++ and NesC). Lua was the next most common language used, with 4 papers.

4.3 Coroutine implementation

To implement coroutines, 27 papers (77%) used a native method, i.e. avoiding techniques that required a new or changed tool-chain. In native implementations, ~~macros (~~13 papers ~~) and libraries~~



(a) RQ1a - Programming languages used (b) RQ1b - Coroutine implementation

Fig. 2. Language outcomes

~~(~~employed macros (of which 7 were based on Duff's device) and 4 ~~papers) were common~~used libraries; in 3 papers ([Cohen et al. 2007; Kalebe et al. 2017; Yu et al. 2008]) the C setjmp/longjmp language device was used.

Several studies extended the ~~tool-chain~~ tool chain or created a new tool. 2 papers contributed new languages [Evers et al. 2007; Jahier 2016], and one paper [Niebert and Caralp 2014] provided a set of language extensions. 2 papers ~~[Jahier 2016; Karpinski and Cahill 2007]~~ employed a transpiler ~~, one~~ to translate from one language to another - one from Lustre to OCaml [Jahier 2016] and one from a synchronous extension of C to standard C [Karpinski and Cahill 2007]. One paper [Fritzsche and Siemers 2010] used a precompiler, and one paper [Jääskeläinen et al. 2008] provided a new compiler optimisation phase.

Two studies called out to another language to implement the coroutines: one [Park et al. 2015], written in the Lua language [Moura et al. 2004], directly manipulated the hosting environment through the C API; another [Khezri et al. 2008] used non-portable assembly language. The results are summarised in Figure 2b.

4.4 Operating system

~~RQ1c – Operating systems used in selected studies using C-like languages~~

Of the 26 instances studied that were written in C, C++ or NesC, 13 (50%) used (or extended) a widely-known embedded operating system (Contiki [Dunkels et al. 2004], TinyOS [Levis et al. 2005] or FreeRTOS [Barry 2018]) and 9 (35%) used a unique operating system, or one that was generated for each application, as shown in Figure 3. There was not enough information in the papers themselves to judge how many of these 9 papers could be considered 'bare-metal'.

4.5 Memory

Figure 4a shows the ROM and RAM sizes of the selected platforms, using logarithmic scales. As observed in RQ2a, there were many systems with low RAM sizes: the median value was 10 kb. There was a positive correlation (r=0.64) between ROM size and RAM size.

4.6 Processors

Only 45% (13 out of 29) of the CPUs that were identified were 32-bit processors: 9 were 8-bit and 7 were 16-bit. The fact that 55% (16 out of 29 studies) used 8- and 16-bit devices indicates that coroutines are applicable to very constrained platforms.

It is also notable that within the 8-bit segment, all but one were of the megaAVR family; among 16-bit processors 5 out of 7 used the TI MSP430 architecture. Within the 32-bit segment the picture

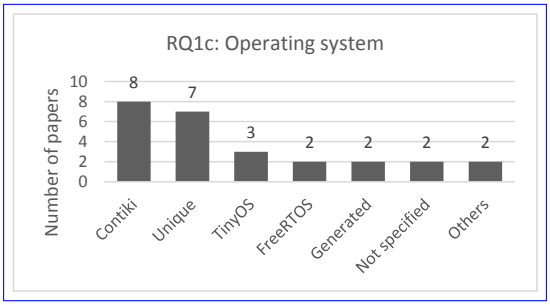
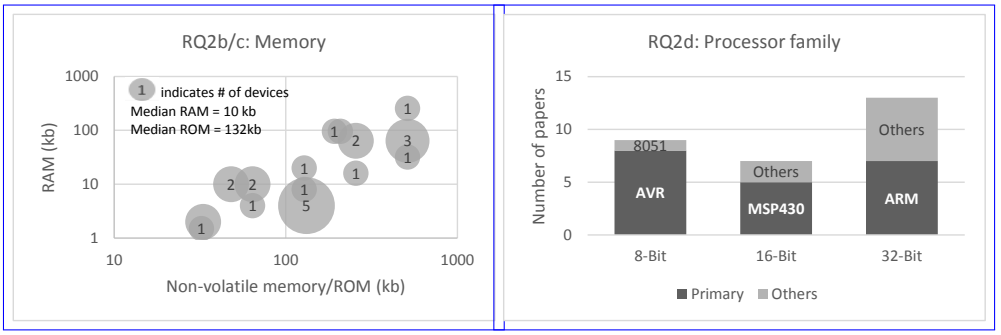


Fig. 3. ~~RQ2b/c~~ RQ1c - ~~Memory~~Operating systems used in selected studies using C-like languages ~~RQ2d - Processor families arranged by bits~~ ~~Hardware outcomes~~



(a) RQ2b/c - Memory (b) RQ2d - Processor families arranged by bits

Fig. 4. Hardware outcomes

was less clear-cut: just over half used the ARM architecture. These types of microcontrollers are widespread in IoT and embedded systems [AspenCore Global Media 2017]. These results are summarised in Figure 4b. Full details are in the supplementary materials.

4.7 Use cases

~~RQ3 - Use cases~~ ~~RQ4 - Intended benefits~~ ~~Usage outcomes~~

The four most common use cases were concurrency (49% of papers), network communication (43%), sensor readings (26%) and data flow (20%), as illustrated in Figure 5a. It is notable that all four of these use cases are often considered to present difficulty or complexity for programmers. (See the supplementary materials for details of use cases and their classifications.)

These use cases are common across many platforms, and not just resource-constrained devices. Syntax designed for desktop systems is likely to handle these cases relatively well. Contrasting these use cases with those found in desktop development, we observe that user interfaces (a strong driver of coroutines in desktop and portable system development) are absent and that sensor readings (a rare requirement in desktop systems) are prominent.

4.8 Intended benefits

Of the intended benefits the most common classifications were (i) code style and simplicity (34%), (ii) scheduling (26%) and (iii) efficiency (23%), as summarized in Figure 5b. (The supplementary materials contain details of the classifications of benefits.)

(a) RQ3 - Use cases (b) RQ4 - Intended benefits

Fig. 5. Usage outcomes

We have observed that split-phase programming leads to error-prone, hard-to-maintain code; it is therefore unsurprising that code style and simplicity leads the list.

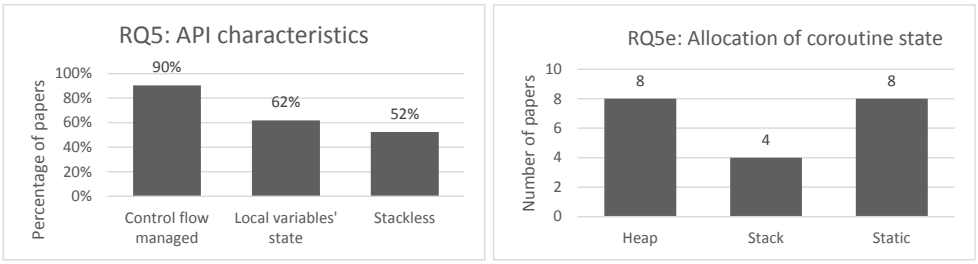
However, the popularity of scheduling as a benefit of a coroutine implementation is not mirrored in mainstream desktop programming, and it may therefore not figure high in the priorities of the C++ language specification process. Coroutines provide a tool with which to build schedulers, and many embedded software applications must provide their own scheduler, either because of the special requirements of the device [Inam et al. 2011; Park et al. 2015; Susilo et al. 2009] or to minimize code size by providing only the minimum requirements.

The high incidence of efficiency as an intended benefit also reflects the latency constraints of embedded systems.

4.9 Application programming interface

Of the 35 studies analysed, 21 discussed an implementation of coroutines: the questionnaire results for RQ5 are listed in the supplementary materials.

The API questions (RQ5b-e) could not in all cases be answered directly from inspection of the ~~paper~~papers. In these cases, unless the answer could be found in the supplementary materials, linked source code, or was well-known to the researchers, the question was answered 'Unknown'. In some cases the source code referenced by the paper was no longer available.



(a) RQ5 - API characteristics (b) RQ5e - Allocation of coroutine state

Fig. 6. API outcomes

Figure 6a summarises the basic API characteristics for those studies which examined an implementation of coroutines. The overwhelming majority (89%) of implementations managed control flow on behalf of the programmer; more than two-thirds managed the state of local variables. The

outcome with regard to stackless and stackful implementations was more balanced: 11 stackless versus 8 stackful.

We have observed that managed, deterministic use of memory is a common requirement for embedded systems: in Figure 6b we see that over a third of papers (8 of 21) supported the allocation of coroutine state on the heap, which is not appropriate for embedded systems, and 4 used the stack, which may not be appropriate if the state size is large or of a size unknown at compile time.

5 ANALYSIS AND DISCUSSION

5.1 Analysis of API design

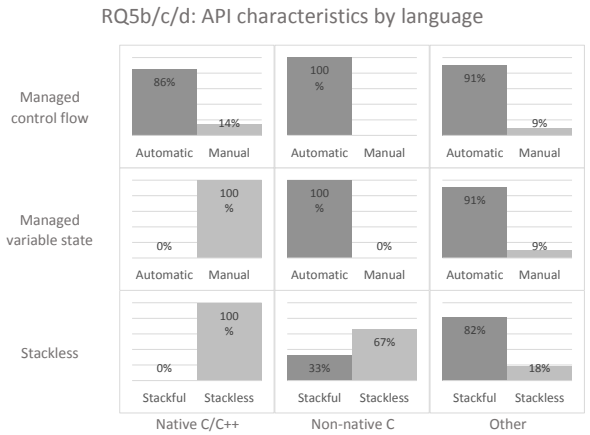


Fig. 7. RQ5b/c/d - API characteristics by language

Figure 7 examines the API characteristics of the various implementations, grouped into (i) native C/C++, (ii) non-native C and (iii) languages other than C, where non-native C refers to language extensions, transpilers, or tools that otherwise extend the C compiler toolchain. The results for languages other than C and for non-native C implementations are interesting because they may reveal what the language designers and implementers considered to be desirable characteristics. (In each case the percentage shown is a fraction of the unique implementations inspected; it is not necessarily representative of the population at large.)

This paper has suggested that the management of control flow on behalf of the programmer (RQ5b) is a desirable feature of programming languages on resource-constrained platforms. The results in Figure 7 appear to support this claim. All non-native C and almost all non-C implementations provide support for managing control flow. (The only exception is ~~[Motika and von Hanxleden 2015]~~ found in the work of Motika and von Hanxleden, a pattern whose code is primarily designed as a target for code generators [Motika and von Hanxleden 2015].) Additionally, 86% of the native C cases were able to provide this feature, primarily through macros.

The management of the state of a coroutine's local variables (RQ5c) has also been proposed as a desirable characteristic. Once again, all non-native C and almost all non-C implementations provide support for this feature. None of the native C implementations were able to provide it, as a consequence of the language's limitations.

None of the native C implementations and only ~~1~~ one of the non-native C implementations were stackful; in contrast 82% of the non-C implementations were stackful. It could be argued that this split indicates that, while stackfulness is a desirable feature for language designers ~~. However~~in

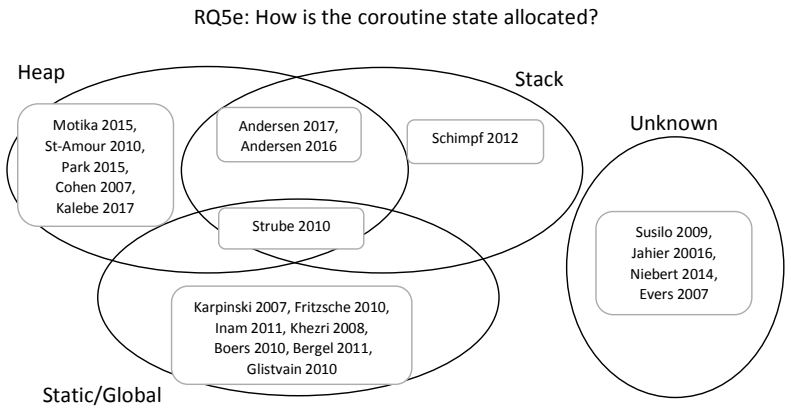


Fig. 8. RQ5e - How is the coroutine state allocated?

general, it is less desirable for 'C' developers. Our interpretation is that, because of the perceived costs of stackfulness in terms of memory and speed, there remains strong support in the C/C++ developer community for stackless coroutines [Dunkels et al. 2006].

The allocation of coroutine state is an important feature of the design with regard to its effect on resource-constrained platforms, since it must be controlled carefully if the design is to offer predictable and safe behaviour. Of the 16 implementations where we were able to determine the allocation method~~was determinable~~, nearly a third used an object or structure to store the state. 44% (7 instances) required that the state be allocated in static (global) memory; 1 used only the stack, and 3 offered flexibility as regards the location.

5 studies ([Cohen et al. 2007; Kalebe et al. 2017; Motika and von Hanxleden 2015; Park et al. 2015; St-Amour and Feeley 2010]) required that the state be stored on the heap (i.e. in dynamically allocated memory space). Of these, 3 were in languages that required such a strategy (Java, Scheme and Lua) and only two ([Cohen et al. 2007; Kalebe et al. 2017]) used a C-based language (NesC or C++). In the case of [Cohen et al. 2007], each coroutine stack of 256 bytes was allocated on the heap. However, the total number of coroutine stacks required was known in advance, and a safe allocation strategy was therefore feasible. Figure 8 summarises these memory strategies.

Given that mainstream C++ programming supports environments where heap memory is generally plentiful, any standard implementation of coroutines in C++ must support dynamic memory allocation for coroutine state storage. However, the special case of ~~constrained-resource~~ resource-constrained platforms, including embedded systems, requires that the developer have the option to use stack memory or global static memory, and that they have full control over which is used on each instantiation. An implementation that supports all three strategies, and allows control over which is used, is therefore desirable.

5.2 Research gaps

Focussing specifically on the studies that describe an implementation, we have analysed the ~~research~~ issues that were addressed by the research in order to identify gaps, as shown in Table 3. Most studies considered the memory and computational cost of the coroutine system, whereas fewer authors addressed interoperability with legacy code. The issue of predictable memory usage by coroutines is particularly important for embedded systems; although 10 of the 21 papers offered

Table 3. Summary of research gaps

Paper	Language was C/C++	Predictable memory usage	Integrates with other language features	Maintainability & readability	Labour cost of implementing the infrastructure	Memory and processing cost of infrastructure	Continued use of legacy code
Cohen et al. 2007		✓✓	✓✓	✓✓	✓✓	✓✓	
Evers et al. 2007		✓				✓✓	
Karpinski et al. 2007		✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
Kumar et al. 2007	✓✓	✓				✓✓	✓✓
Khezri et al. 2008			✓✓			✓	✓✓
Susilo et al. 2009	✓✓					✓✓	
Boers et al. 2010	✓✓		✓			✓✓	
Fritzsche et al. 2010	✓✓		✓✓				
Glistvain et al. 2010	✓✓	✓✓			✓✓	✓✓	
St-Amour et al. 2010			✓✓	✓✓	✓✓	✓✓	
Strube et al. 2010	✓✓	✓✓	✓				
Bergel et al. 2011		✓✓		✓✓		✓✓	
Inam et al. 2011	✓✓				✓	✓✓	✓
Schimpf 2012	✓✓		✓	✓	✓✓	✓✓	✓
Niebert et al. 2014	✓✓	✓✓				✓✓	
Motika et al. 2015			✓✓	✓		✓✓	
Park et al. 2015		✓✓	✓✓	✓✓		✓✓	✓
Andersen et al. 2016		✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
Jahier 2016		✓✓	✓✓		✓✓		✓✓
Andersen et al. 2017		✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
Kalebe et al. 2017	✓✓		✓✓	✓✓	✓✓		
Ideal outcome	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓

Key: ✓✓- The issue is considered and resolved. ✓- The issue is addressed but not resolved.
Blank – The issue is not present.

a solution, none of these solutions will apply to a C++ native solution which also handles local variable state.

We conclude that a research gap remains with regard to the study of standard C++ as an appropriate language for the development of asynchronous programs on resource-constrained devices.

5.3 Repeatability of search results

We found that when the IEEE Xplore database search was repeated 11 months after the original search, the new results were not, as they were expected to be, a superset of the original results.

Of the original 144 papers found in October 2017, only 87 (60%) appeared in the search results in September 2018. Further, of the 32 new results, only 16 were papers published since 2015: the other half were published before 2015. We conclude that the search methodology of the IEEE database has changed in the interim, and this raises a question over the use of this database for systematic surveys. This problem was not found for the other on-line databases used.

5.4 Discussion

The majority of selected papers used the C programming language; while the coroutines proposal [ISO/IEC 2017] is for C++, the use of C++ for these projects would not necessarily require significant programming changes.

Over a quarter of the papers used the Contiki [Dunkels et al. 2004] operating system, which provides coroutine support through Protothreads [Dunkels et al. 2006], which rely on Duff's device. Given the problems, discussed in ~~section~~ Section 2.3, that are associated with using this device, this common use of Protothreads indicates a widespread need for the facilities provided by a coroutine-like solution.

More than half (55%) of the studies used 16-bit or 8-bit processors. Support for these platforms on leading C++ compilers is currently limited; this will need to be addressed before C++ coroutines can be applied to the smallest platforms.

As expected [AspenCore Global Media 2017; Skerrett 2017], code style or simplicity was the leading desired benefit of the language feature implementation. The second most common benefit was a basis for a scheduler: this is not commonly a perceived benefit of coroutines on mainstream platforms, and this difference warrants further study.

The coding of three common use cases - communications, data-flow and sensor readings - present particular difficulties on constrained-resource devices, because these problems require the use of split-phase programming. Each of these problems could be addressed using programming patterns enabled by coroutines: async/await and generator. These patterns would enable a direct programming style that is likely to reduce development effort and the incidence of errors. The high incidence of these use cases in our survey indicate that they represent an important and worthwhile target for further study.

Our survey indicates that multiple studies exist that require a coroutine-based facility for concurrent programming on resource-constrained devices, establishing that a demand exists at this end of the spectrum, not merely on high performance platforms. We noted in ~~section~~ Section 5.1 that, where the language allowed fine-tuned management of memory allocation, dynamic allocation of memory was avoided for coroutine state and stack. We can conclude that avoiding heap memory is a requirement for the small devices that formed the bulk of the target platforms.

While 82% of non-C implementations were stackful, only one of the C implementations was stackful~~: it can be concluded~~. We observe that when a language is designed from the ground up to support coroutines, then a stackful implementation is common. On the other hand, such a feature is difficult in C while preserving both backward compatibility and acceptable memory usage. We conclude that a stackless implementation is important to C programmers, and this reflects the scarcity of memory resources on the platforms under consideration.

None of the works studied utilize a coroutine implementation for C or C++ that provides managed variable state and that is designed specifically for an event-driven environment on a resource-constrained platform. We therefore conclude that this represents a significant research gap, and that further work towards such an implementation is warranted.

Although this survey found 20 papers that used C and only 2 that used C++, there is evidence that a migration from C to C++ on resource-constrained devices is occurring [AspenCore Global Media 2017]. Developers may also be motivated to make the switch from C to C++ to gain access to a clean

implementation of coroutines to support the async/await and generator patterns and lightweight scheduling, as provided by the proposed C++20 standard [ISO/IEC 2017]. However, this will require language and library support appropriate for resourced constrained devices. Implementers should consider ways to avoid two of the C++ features considered dangerous by Goldthwaite [Goldthwaite 2006]: dynamic memory allocation and exceptions. It would be particularly useful to establish whether the proposed C++ coroutine implementations can offer deterministic memory utilisation, known at compile time: this would make it possible to avoid dynamic memory allocation.

We have seen that various specialized solutions have been applied to the problem of providing direct programming style for split-phase code on embedded systems, including Protothreads, precompilers, language extensions, post-compilation optimization phases and non-portable code libraries. It is clear that, on the one hand, coroutines offer many benefits for software development on these devices but, on the other hand, the implementation is challenging. By contrast, implementing coroutines in C++ on mainstream enterprise systems is relatively straightforward, because there are resources to spare, including memory, operating system facilities and standard libraries. While adapting coroutines for resource constrained devices may be more difficult, it offers greater benefits, because the use cases are such a good fit for embedded systems, including the low-cost, low-power scheduling, communications and sensor management that are often needed by Internet of Things applications.

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

This paper has analysed the current academic body of work regarding the use of asynchronous programming techniques in embedded systems. We conclude that there exists significant demand for these facilities. We argue that embedded systems must be considered as part of the debate around the standardisation of coroutines in C++. The C++ proposals provide an opportunity to improve the software engineering of embedded systems but only if the language facilities are useful in an extremely resource-constrained environment.

6.2 Future work

Future work could include the following:

- Investigate whether the N4680 proposal can provide deterministic memory use, with full control over the detail of allocation and, if not, what changes would need to be made to the specification. Similarly, test whether the current implementations (Microsoft C++ 14.1 [Microsoft Corporation 2018] and LLVM 6.0.1 [LLVM Project 2018]) provide this determinism and control.
- Investigate whether the N4680 proposal and its implementations can work effectively in an event-driven environment on a resource-constrained platform, with and without a real-time operating system.
- Study the memory and performance costs of the current N4680 implementations on resource-constrained platforms with minimal or no operating system support.

ACKNOWLEDGMENTS

The first author is supported by an Australian Government Research Training Program (RTP) Scholarship.

REFERENCES

Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. 2015. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys and Tutorials* 17, 4 (2015), 2347–2376. https://doi.org/10.1109/COMST.2015.2444095 arXiv:arXiv:1011.1669v3

Mariano Alvira and Taylor Barton. 2013. *Small and Inexpensive Single-Board Computer for Autonomous Sailboat Control*. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 105–116. https://doi.org/10.1007/978-3-642-33084-1_10

Sidharta Andalam, Partha S. Roop, Alain Girault, and Claus Traulsen. 2014. A Predictable Framework for Safety-Critical Embedded Systems. *IEEE Trans. Comput.* 63, 7 (2014), 1600–1612. https://doi.org/10.1109/TC.2013.28

Michael P. Andersen, Gabe Fierro, and David E. Culler. 2016. System Design for a Synergistic, Low Power Mote/BLE Embedded Platform. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, Vienna, Austria, 1–12. https://doi.org/10.1109/IPSN.2016.7460722

Michael P. Andersen, Gabe Fierro, and David E. Culler. 2017. Enabling synergy in IoT: Platform to service and beyond. *Journal of Network and Computer Applications* 81, October 2016 (2017), 96–110. https://doi.org/10.1016/j.jnca.2016.10.017

AspenCore Global Media. 2017. 2017 Embedded Markets Study. http://m.eet.com/media/1246048/2017-embedded-market-study.pdf

Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks* 54, 15 (2010), 2787–2805. https://doi.org/10.1016/j.comnet.2010.05.010 arXiv:arXiv:1011.1669v3

Richard Barry. 2018. The FreeRTOS Kernel. https://www.freertos.org/

Alexandre Bergel, William Harrison, Vinny Cahill, and Siobhán Clarke. 2011. FlowTalk: Language Support for Long-Latency Operations in Embedded Devices. *IEEE Transactions on Software Engineering* 37, 4 (2011), 526–543. https://doi.org/10.1109/TSE.2010.66

Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause 'n' Play: Formalizing Asynchronous C sharp. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–257. https://doi.org/10.1007/978-3-642-31057-7_12

Nicholas M. Boers, Paweł Gburzyński, Ioanis Nikolaidis, and Włodek Olesiński. 2010. Developing wireless sensor network applications in a virtual environment. *Telecommunication Systems* 45, 2-3 (2010), 165–176. https://doi.org/10.1007/s11235-009-9246-x

Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software* 80, 4 (2007), 571–583. https://doi.org/10.1016/j.jss.2006.07.009

Etienne Brodu, Stéphane Frénot, and Frédéric Oblé. 2015. Toward Automatic Update from Callbacks to Promises. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems - AWeS '15*. ACM Press, New York, New York, USA, 1–8. https://doi.org/10.1145/2749215.2749216

David L. Clark. 2009. Powering intelligent instruments with Lua scripting. In *2009 IEEE AUTOTESTCON*. IEEE, 101–106. https://doi.org/10.1109/AUTEST.2009.5314042

Marcelo Cohen, Thiago Ponte, Silvana Rossetto, and Noemi Rodriguez. 2007. Using Coroutines for RPC in Sensor Networks. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8. https://doi.org/10.1109/IPDPS.2007.370458

Melvin E Conway. 1963. Design of a Separable Transition-diagram Compiler. *Commun. ACM* 6, 7 (jul 1963), 396–408. https://doi.org/10.1145/366663.366704

Tom Duff. 1988. Duff's Device. https://groups.google.com/forum/#!original/comp.lang.c/sWJtHn6sVps/ESTEdYEpeLsJ

Adam Dunkels. 2005. About protothreads. http://dunkels.com/adam/pt/about.html

A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*. IEEE (Comput. Soc.), 455–462. https://doi.org/10.1109/LCN.2004.38

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. ACM Press, New York, New York, USA, 29—-42. https://doi.org/10.1145/1182807.1182811

Caglar Durmaz, Moharram Challenger, Orhan Dagdeviren, and Geylani Kardas. 2017. Modelling Contiki-Based IoT Systems. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017) (OpenAccess Series in Informatics (OASIcs))*, Ricardo Queirós, Mário Pinto, Alberto Simões, José Paulo Leal, and Maria João Varanda (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1—-5:13. https://doi.org/10.4230/OASIcs.SLATE.2017.5

ECMA. 2017. ECMAScript Latest Draft (ECMA-262) Async Function Definitions. https://tc39.github.io/ecma262/#sec-async-function-definitions

Jonathan Edwards. 2009. Coherent reaction. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*. ACM Press, New York, New York, USA, 925. https://doi.org/10.1145/1639950.1640058

Atis Elsts, George Oikonomou, Xenofon Fafoutis, and Robert Piechocki. 2017. Internet of Things for smart homes: Lessons learned from the SPHERE case study. In *2017 Global Internet of Things Summit (GIoTS)*. 1–6. https://doi.org/10.1109/GIOTS.2017.8016226

Ralf S Engelschall. 2000. Portable Multithreading-The Signal Stack Trick for User-Space Thread Creation. In *USENIX Annual Technical Conference, General Track*. https://dl.acm.org/citation.cfm?id=1267744

L Evers, P J M Havinga, J Kuper, M E M Lijding, and N Meratnia. 2007. SensorScheme: Supply chain management automation using Wireless Sensor Networks. In *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*. 448–455. https://doi.org/10.1109/EFTA.2007.4416802

Rene Fritzsche and Christian Siemers. 2010. Scheduling of Time Enhanced C (Tec). In *2010 World Automation Congress*. 1–6.

David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation - PLDI '03*. ACM Press, New York, New York, USA, 1—-11. https://doi.org/10.1145/781131.781133

Damien George. 2014. MicroPython - Python for microcontrollers. http://micropython.org/

Damien P. George and Paul Sokolovsky. 2014. General information about the ESP8266 port — MicroPython 1.9.4 documentation. http://docs.micropython.org/en/latest/esp8266/esp8266/general.html

R. Glistvain and M. Aboelaze. 2010. Romantiki OS - A single stack multitasking operating system for resource limited embedded devices. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*. 1–8. https://ieeexplore.ieee.org/document/5461735

Lois Goldthwaite. 2006. Technical report on C++ performance. *ISO/IEC PDTR* 18015 (2006).

Nat Goodspeed. 2014. *Stackful Coroutines and Stackless Resumable Functions*. Technical Report. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4232.pdf

Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems* 29, 7 (2013), 1645–1660. https://doi.org/10.1016/j.future.2013.01.010 arXiv:1207.0203

R K Gupta, C N Coelho Jr., and G De Micheli. 1992. Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC '92)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 225–230. http://dl.acm.org/citation.cfm?id=113938.149413

Philipp Haller and Jason Zaugg. 2016. SIP-22 - Async - Scala Documentation. http://docs.scala-lang.org/sips/pending/async.html

Ralph Hempel. 2008. Porting Lua to a microcontroller. *Lua Programming Gems* (2008).

Dominic Herity. 2015. Modern C++ in embedded systems – Part 1: Myth and Reality | Embedded. https://www.embedded.com/design/programming-languages-and-tools/4438660/1/Modern-C--in-embedded-systems---Part-1--Myth-and-Reality

R Inam, J Mäki-Turja, M Sjödin, S M H Ashjaei, and S Afshar. 2011. Support for hierarchical scheduling in FreeRTOS. In *ETFA2011*. 1–10. https://doi.org/10.1109/ETFA.2011.6059016

ISO/IEC. 2017. N4680 Programming Languages - C++ Extensions for Library Fundamentals. (2017). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4680.pdf

Pekka Jääskeläinen, Pertti Kellomäki, Jarmo Takala, Heikki Kultala, and Mikael Lepistö. 2008. Reducing Context Switch Overhead with Compiler-Assisted Threading. In *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Vol. 2. 461–466. https://doi.org/10.1109/EUC.2008.181

Erwan Jahier. 2016. RDBG: A Reactive Programs Extensible Debugger. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES '16)*, Sander Stuijk (Ed.). ACM, New York, NY, USA, 116–125. https://doi.org/10.1145/2906363.2906372

Rubem Kalebe, Gustavo Girao, and Itamir Filho. 2017. A library for scheduling lightweight threads in Internet of Things microcontrollers. In *2017 International Conference on Computing Networking and Informatics (ICCNI)*. IEEE, 1–7. https://doi.org/10.1109/ICCNI.2017.8123793

Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2013. An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications (DYLA '13)*. ACM, New York, NY, USA, 3:1—-3:9. https://doi.org/10.1145/2489798.2489802

Marcin Karpinski and Vinny Cahill. 2007. High-Level Application Development is Realistic for Wireless Sensor Networks. In *2007 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. 610–619. https://doi.org/10.1109/SAHCN.2007.4292873

Meysam Khezri, Mehdi Agha Sarram, and Fazlollah Adibniya. 2008. Simplifying Concurrent Programming of Networked Embedded Systems. In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. 993–998. https://doi.org/10.1109/ISPA.2008.138

Barbara Kitchenham. 2004. *Procedures for performing systematic reviews*. Technical Report TR/SE-0401. 28 pages. https://doi.org/10.1.1.122.3308

Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing Systematic Literature reviews in Software Engineering Version 2.3.* Technical Report. arXiv:1304.1186

Barbara A. Kitchenham, David Budgen, and O. Pearl Brereton. 2011. Using mapping studies as the basis for further research - A participant-observer case study. *Information and Software Technology* 53, 6 (2011), 638–651. https://doi.org/10.1016/j.infsof.2010.12.011

Donald E Knuth. 1997. The Art of Computer Programming, Vol. 1: Fundamental Algorithms (3rd. ed.). (1997).

Patrick Kugler, Philipp Nordhus, and Bjoern Eskofier. 2013. Shimmer, Cooja and Contiki: A new toolset for the simulation of on-node signal processing algorithms. In *2013 IEEE International Conference on Body Sensor Networks*. 1–6. https://doi.org/10.1109/BSN.2013.6575497

Nagendra J. Kumar, Vasanth Asokan, Siddhartha Shivshankar, and Alexander G. Dean. 2007. Efficient Software Implementation of Embedded Communication Protocol Controllers Using Asynchronous Software Thread Integration with Time- and Space-efficient Procedure Calls. *ACM Transactions on Embedded Computing Systems* 6, 1 (feb 2007). https://doi.org/10.1145/1210268.1210270

Philip Levis and David Culler. 2002. Mate : A Tiny Virtual Machine for Sensor Networks. *ACM SIGPLAN Notices* 37, 10 (2002), 85–95. https://doi.org/10.1145/605397.605407

P Levis, S Madden, J Polastre, R Szewczyk, K Whitehouse, A Woo, D Gay, J Hill, M Welsh, E Brewer, and D Culler. 2005. *TinyOS: An Operating System for Sensor Networks.* Springer Berlin Heidelberg, Berlin, Heidelberg, 115–148. https://doi.org/10.1007/3-540-27139-2_7

B. Liskov and L. Shrira. 1988. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices* 23, 7 (1988), 260–267. https://doi.org/10.1145/960116.54016

Weichen Liu, Jiang Xu, Jogesh K. Muppala, Wei Zhang, Xiaowen Wu, Yaoyao Ye, and Wei Zhang. 2011. Coroutine-Based Synthesis of Efficient Embedded Software From SystemC Models. *IEEE Embedded Systems Letters* 3, 1 (2011), 46–49. https://doi.org/10.1109/LES.2011.2112634

LLVM Project. 2018. Download LLVM releases. http://releases.llvm.org/

Daniel Lohmann, Olaf Spinczyk, Wanja Hofer, and Wolfgang Schröder-Preikschat. 2012. *The Aspect-Aware Design and Implementation of the CiAO Operating-System Family.* Springer, Berlin, Heidelberg, 168–215. https://doi.org/10.1007/978-3-642-35551-6_5

Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017). https://doi.org/10.1145/3133910

James Manyika, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. 2015. The Internet of Things: Mapping the value beyond the hype. *McKinsey Global Institute* June (2015), 144. https://www.mckinsey.com/~/media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/The%20Internet%20of%20Things%20The%20value%20of%20digitizing%20the%20physical%20world/The-Internet-of-things-Mapping-the-value-beyond-the-hype.ashx

Christopher D Marlin. 1979. *Coroutines: A Programming Methodology, a Language Design and an Implementation.* Number 95 in Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.

Erik Meijer. 2010. Reactive extensions (Rx): Curing Your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming on - CUFP '10*. ACM Press, New York, New York, USA, 1. https://doi.org/10.1145/1900160.1900173

Microsoft Corporation. 2018. The latest supported Visual C++ downloads. https://support.microsoft.com/en-au/help/2977003/the-latest-supported-visual-c-downloads

Eric Mittelette. 2015. Coroutines in Visual Studio 2015 – Update 1 - Visual C++ Team Blog. https://blogs.msdn.microsoft.com/vcblog/2015/11/30/coroutines-in-visual-studio-2015-update-1/

Christian Motika and Reinhard von Hanxleden. 2015. Light-weight Synchronous Java (SJL): An approach for programming deterministic reactive systems with Java. *Computing* 97, 3 (2015), 281–307. https://doi.org/10.1007/s00607-014-0416-7

Ana Lúcia De Moura, Noemi Rodriguez, and Roberto Ierusalimschy. 2004. Coroutines in Lua. *Journal of Universal Computer Science* 10, 7 (2004), 910–925. https://doi.org/10.3217/jucs-010-07-0910

Peter Niebert and Mathieu Caralp. 2014. *Cellular Programming.* Springer International Publishing, Cham, 11–22. https://doi.org/10.1007/978-3-319-13749-0_2

Uzair A. Noman, Behailu Negash, Amir M. Rahmani, Pasi Liljeberg, and Hannu Tenhunen. 2017. From threads to events: Adapting a lightweight middleware for Contiki OS. In *2017 14th IEEE Annual Consumer Communications and Networking Conference (CCNC)*. 486–491. https://doi.org/10.1109/CCNC.2017.7983156

Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. 2014. A Study and Toolkit for Asynchronous Programming in C#. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014.* ACM Press, New York, New York, USA, 1117–1127. https://doi.org/10.1145/2568225.2568309

Frank Oldewurtel, Janne Riihijärvi, Krisakorn Rerkrai, and Petri Mähönen. 2009. The RUNES Architecture for Reconfigurable Embedded and Sensor Networks. In *2009 Third International Conference on Sensor Technologies and Applications.* 109–116.

https://doi.org/10.1109/SENSORCOMM.2009.26

Jonathan Paisley and Joseph Sventek. 2006. Real-time Detection of Grid Bulk Transfer Traffic. In *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*. 66–72. https://doi.org/10.1109/NOMS.2006.1687539

Sihyeong Park, Hyungshin Kim, Soo Yeong Kang, Cheol Hea Koo, and Hyunwoo Joe. 2015. Lua-Based Virtual Machine Platform for Spacecraft On-Board Control Software. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*. 44–51. https://doi.org/10.1109/EUC.2015.21

Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. 2008. Systematic Mapping Studies in Software Engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. BCS Learning & Development Ltd., 68—-77.

Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (2015), 1–18. https://doi.org/10.1016/j.infsof.2015.03.007 arXiv:arXiv:1011.1669v3

Till Riedel, Nicolaie Fantana, Adrian Genaid, Dimitar Yordanov, Hedda R Schmidtke, and Michael Beigl. 2010. Using web service gateways and code generation for sustainable IoT system development. In *2010 Internet of Things (IOT)*. 1–8. https://doi.org/10.1109/IOT.2010.5678449

Torvald Riegel. 2015. P0073R0 - On unifying the coroutines and resumable functions proposals. (2015). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0073r0.pdf

Geoff Romer, James Dennett, and Chandler Carruth. 2018. P1063R0 - Core Coroutines Making coroutines simpler, faster, and more general. (2018). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1063r0.pdf

Silvana Rossetto and Noemi Rodriguez. 2006. A cooperative multitasking model for networked sensors. *Proceedings - International Conference on Distributed Computing Systems* (2006). https://doi.org/10.1109/ICDCSW.2006.5

Paul H Schimpf. 2012. Modified Protothreads for Embedded Systems. *Journal of Computing Sciences in Colleges* 28, 1 (2012), 177–184. http://dl.acm.org/citation.cfm?id=2379703.2379738

S. Sicari, A. Rizzardi, L.A. Grieco, and A. Coen-Porisini. 2015. Security, privacy and trust in Internet of Things: The road ahead. *Computer Networks* 76 (jan 2015), 146–164. https://doi.org/10.1016/j.comnet.2014.11.008 arXiv:1404.7799

Ian Skerrett. 2017. IoT Developer Trends 2017 Edition. https://ianskerrett.wordpress.com/2017/04/19/iot-developer-trends-2017-edition/

Vincent St-Amour and Marc Feeley. 2010. *PICOBIT: A Compact Scheme System for Microcontrollers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-16478-1_1

Bjarne Stroustrup. 1986. An Overview of C++. In *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming (OOPWORK '86)*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/323779.323736

M Strube, M Daum, R Kapitza, F Villanueva, and F Dressler. 2010. Dynamic operator replacement in sensor networks. In *The 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2010)*. 748–750. https://doi.org/10.1109/MASS.2010.5663821

E. Susilo, P. Valdastri, A. Menciassi, and P. Dario. 2009. A miniaturized wireless control platform for robotic capsular endoscopy using advanced pseudokernel approach. *Sensors and Actuators, A: Physical* 156, 1 (2009), 49–58. https://doi.org/10.1016/j.sna.2009.03.036

Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The Fsharp asynchronous programming model. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6539 LNCS (2011), 175–189. https://doi.org/10.1007/978-3-642-18378-2_15

Simon Tatham. 2000. Coroutines in C. https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

The Python Software Foundation. 2018. 9. Classes — Python 3.6.4 documentation. https://docs.python.org/3/tutorial/classes.html#generators

C. Tismer. 2000. Continuations and stackless Python. *Proceedings of the 8th International Python Conference* (2000), 2000–01. https://svn.python.org/www/trunk/pydotorg/workshops/2000-01/proceedings/papers/tismers/spcpaper.doc

Christian Tismer. 2018. About Stackless. https://github.com/stackless-dev/stackless/wiki

Rob van der Meulen. 2017. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. http://www.gartner.com/newsroom/id/3598917

Guido van Rossum and Phillip J. Eby. 2005. PEP 342 – Coroutines via Enhanced Generators. (2005). https://www.python.org/dev/peps/pep-0342/

Reinhard von Hanxleden. 2009. SyncCharts in C: A Proposal for Light-weight, Deterministic Concurrency. In *Proceedings of the Seventh ACM International Conference on Embedded Software*. ACM, New York, NY, USA, 225–234. https://doi.org/10.1145/1629335.1629366

Rolf H Weber. 2015. Internet of things: Privacy issues revisited. *Computer Law & Security Review: The International Journal of Technology Law and Practice* 31 (2015), 618–627. https://doi.org/10.1016/j.clsr.2015.07.002

Min Yu, SiJi Xiahou, and XinYu Li. 2008. A Survey of Studying on Task Scheduling Mechanism for TinyOS. *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing* (2008), 1–4. https://doi.org/10.

1109/WiCom.2008.960