Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems

Adam Dunkels[†], Oliver Schmidt, Thiemo Voigt[†], Muneeb Ali[‡*]
[†]Swedish Institute of Computer Science, Box 1263, SE-16429 Kista, Sweden
[‡]TU Delft, Mekelweg 4, 2628 CD Delft,The Netherlands
adam@sics.se, oliver@jantzer-schmidt.de, thiemo@sics.se, m.ali@tudelft.nl

Abstract

Event-driven programming is a popular model for writing programs for tiny embedded systems and sensor network nodes. While event-driven programming can keep the memory overhead down, it enforces a state machine programming style which makes many programs difficult to write, maintain, and debug. We present a novel programming abstraction called protothreads that makes it possible to write event-driven programs in a thread-like style, with a memory overhead of only two bytes per protothread. We show that protothreads significantly reduce the complexity of a number of widely used programs previously written with event-driven state machines. For the examined programs the majority of the state machines could be entirely removed. In the other cases the number of states and transitions was drastically decreased. With protothreads the number of lines of code was reduced by one third. The execution time overhead of protothreads is on the order of a few processor cycles.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Wireless sensor networks, Embedded systems, Threads

1 Introduction

Event-driven programming is a common programming model for memory-constrained embedded systems, including sensor networks. Compared to multi-threaded systems, event-driven systems do not need to allocate memory for per-thread stacks, which leads to lower memory requirements. For this reason, many operating systems for sensor networks, including TinyOS [19], SOS [17], and Contiki [12] are based on an event-driven model. According to Hill et al. [19]: *"In TinyOS, we have chosen an event model so that high levels of concurrency can be handled in a very small amount of space. A stack-based threaded approach would require that stack space be reserved for each execution context."* Event-driven programming is also often used in systems that are too memory-constrained to fit a general-purpose embedded operating system [28].

An event-driven model does not support a blocking wait abstraction. Therefore, programmers of such systems frequently need to use state machines to implement control flow for high-level logic that cannot be expressed as a single event handler. Unlike state machines that are part of a system specification, the control-flow state machines typically have no formal specification, but are created on-the-fly by the programmer. Experience has shown that the need for explicit state machines to manage control flow makes event-driven programming difficult [3, 25, 26, 35]. With the words of Levis et al. [26]: *"This approach is natural for reactive processing and for interfacing with hardware, but complicates sequencing high-level operations, as a logically blocking sequence must be written in a state-machine style."* In addition, popular programming languages for tiny embedded systems such as the C programming language and nesC [15] do not provide any tools to help the programmer manage the implementation of explicit state machines.

In this paper we study how *protothreads*, a novel programming abstraction that provides a conditional blocking wait operation, can be used to reduce the number of explicit state machines in event-driven programs for memory-constrained embedded systems.

The contribution of this paper is that we show that protothreads simplify event-driven programming by reducing the need for explicit state machines. We show that the protothreads mechanism is simple enough that a prototype implementation of the protothreads mechanism can be done using only C language constructs, without any architecture-specific machine code. We have previously presented the ideas behind protothreads in a position paper [13]. In this paper we significantly extend our previous work by refining the protothreads mechanism as well as quantifying and evaluating the utility of protothreads.

To evaluate protothreads, we analyze a number of widely used event-driven programs by rewriting them using pro-

[*]Work done at the Swedish Institute of Computer Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'06, November 1–3, 2006, Boulder, Colorado, USA.
Copyright 2006 ACM 1-59593-343-3/06/0011 ...$5.00

tothreads. We use three metrics to quantify the effect of protothreads: the number of explicit state machines, the number of explicit state transitions, and lines of code. Our measurements show that protothreads reduce all three metrics for all the rewritten programs. For most programs the explicit state machines can be entirely removed. For the other programs protothreads significantly reduce the number of states. Compared to a state machine, the memory overhead of protothreads is a single byte. The memory overhead of protothreads is significantly lower than for traditional multithreading. The execution time overhead of protothreads over a state machine is a few processor cycles.

We do not advocate protothreads as a general replacement for state machines. State machines are a powerful tool for designing, modeling, and analyzing embedded systems. They provide a well-founded formalism that allows reasoning about systems and in some cases can provide proofs of the behavior of the system. There are, however, many cases where protothreads can greatly simplify the program without introducing any appreciable memory overhead. Specifically, we have seen many programs for event-driven systems that are based on informally specified state machines. The state machines for those programs are in many cases only visible in the program code and are difficult to extract from the code.

We originally developed protothreads for managing the complexity of explicit state machines in the event-driven uIP embedded TCP/IP stack [10]. The prototype implementations of protothreads presented in this paper are also used in the Contiki operating system [12] and have been used by at least ten different third-party embedded developers for a range of different embedded devices. Examples include an MPEG decoding module for Internet TV-boxes, wireless sensors, and embedded devices collecting data from charge-coupled devices. The implementations have also been ported by others to C++ [30] and Objective C [23].

The rest of the paper is structured as follows. Section 2 describes protothreads and shows a motivating example. In Section 3 we discuss the memory requirements of protothreads. Section 4 shows how state machines can be replaced with protothreads. Section 5 describes how protothreads are implemented and presents a prototype implementation in the C programming language. In Section 6 we evaluate protothreads, followed by a discussion in Section 7. We review of related work in Section 8. Finally, the paper is concluded in Section 9.

2 Protothreads

Protothreads are a novel programming abstraction that provides a conditional blocking wait statement, PT_WAIT_UNTIL(), that is intended to simplify event-driven programming for memory-constrained embedded systems. The operation takes a conditional statement and blocks the protothread until the statement evaluates to true. If the conditional statement is true the first time the protothread reaches the PT_WAIT_UNTIL() the protothread continues to execute without interruption. The PT_WAIT_UNTIL() condition is evaluated each time the protothread is invoked. The PT_WAIT_UNTIL() condition can be any conditional statement, including complex Boolean expressions.

A protothread is stackless: it does not have a history of function invocations. Instead, all protothreads in a system run on the same stack, which is rewound every time a protothread blocks.

A protothread is driven by repeated calls to the function in which the protothread runs. Because they are stackless, protothreads can only block at the top level of the function. This means that it is not possible for a regular function called from a protothread to block inside the called function - only explicit PT_WAIT_UNTIL() statements can block. The advantage of this is that the programmer always is aware of which statements that potentially may block. Nevertheless, it is possible to perform nested blocking by using hierarchical protothreads as described in Section 2.5.

The beginning and the end of a protothread are declared with PT_BEGIN and PT_END statements. Protothread statements, such as the PT_WAIT_UNTIL() statement, must be placed between the PT_BEGIN and PT_END statements. A protothread can exit prematurely with a PT_EXIT statement. Statements outside of the PT_BEGIN and PT_END statements are not part of the protothread and the behavior of such statements are undefined.

Protothreads can be seen as a combination of events and threads. From threads, protothreads have inherited the blocking wait semantics. From events, protothreads have inherited the stacklessness and the low memory overhead. The blocking wait semantics allow linear sequencing of statements in event-driven programs. The main advantage of protothreads over traditional threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a thread's stack might use a large part of the available memory. For example, a thread with a 200 byte stack running on an MS430F149 microcontroller uses almost 10% of the entire RAM. In contrast, the memory overhead of a protothread is as low as two bytes per protothread and no additional stack is needed.

2.1 Scheduling

The protothreads mechanism does not specify any specific method to invoke or schedule a protothread; this is defined by the system using protothreads. If a protothread is run on top of an underlying event-driven system, the protothread is scheduled whenever the event handler containing the protothread is invoked by the event scheduler. For example, application programs running on top of the event-driven uIP TCP/IP stack are invoked both when a TCP/IP event occurs and when the application is periodically polled by the TCP/IP stack. If the application program is implemented as a protothread, this protothread is scheduled every time uIP calls the application program.

In the Contiki operating system, processes are implemented as protothreads running on top of the event-driven Contiki kernel. A process' protothread is invoked whenever the process receives an event. The event may be a message from another process, a timer event, a notification of sensor input, or any other type of event in the system. Processes

```
state: {ON, WAITING, OFF}

radio_wake_eventhandler:
  if (state = ON)
    if (expired(timer))
      timer ← t_sleep
      if (not communication_complete())
        state ← WAITING
        wait_timer ← t_wait_max
      else
        radio_off()
        state ← OFF
  elseif (state = WAITING)
    if (communication_complete() or
        expired(wait_timer))
      state ← OFF
      radio_off()
  elseif (state = OFF)
    if (expired(timer))
      radio_on()
      state ← ON
      timer ← t_awake
```

Figure 1. The radio sleep cycle implemented with events, in pseudocode.

may wait for incoming events using the protothread conditional blocking statements.

The protothreads mechanism does not specify how memory for holding the state of a protothread is managed. As with the scheduling, the system using protothreads decides how memory should be allocated. If the system will run a predetermined amount of protothreads, memory for the state of all protothreads can be statically allocated in advance. Memory for the state of a protothread can also be dynamically allocated if the number of protothreads is not known in advance. In Contiki, the memory for the state of a process' protothread is held in the process control block. Typically, a Contiki program statically allocates memory for its process control blocks.

In general, protothreads are reentrant. Multiple protothreads can be running the same piece of code as long as each protothread has its own memory for keeping state.

2.2 Protothreads as Blocking Event Handlers

Protothreads can be seen as blocking event handlers in that protothreads can run on top of an existing event-based kernel, without modifications to the underlying event-driven system. Protothreads running on top of an event-driven system can use the PT_WAIT_UNTIL() statement to block conditionally. The underlying event dispatching system does not need to know whether the event handler is a protothread or a regular event handler.

In general, a protothread-based implementation of a program can act as a drop-in replacement a state machine-based implementation without any modifications to the underlying event dispatching system.

2.3 Example: Hypothetical MAC Protocol

To illustrate how protothreads can be used to replace state machines for event-driven programming, we consider a hypothetical energy-conserving sensor network MAC protocol.

```
radio_wake_protothread:
  PT_BEGIN
  while (true)
    radio_on()
    timer ← t_awake
    PT_WAIT_UNTIL(expired(timer))
    timer ← t_sleep
    if (not communication_complete())
      wait_timer ← t_wait_max
      PT_WAIT_UNTIL(communication_complete() or
                    expired(wait_timer))
    radio_off()
    PT_WAIT_UNTIL(expired(timer))
  PT_END
```

Figure 2. The radio sleep cycle implemented with protothreads, in pseudocode.

One of the tasks for a sensor network MAC protocol is to allow the radio to be turned off as often as possible in order to reduce the overall energy consumption of the device. Many MAC protocols therefore have scheduled sleep cycles when the radio is turned off completely.

The hypothetical MAC protocol used here is similar to the T-MAC protocol [34] and switches the radio on and off at scheduled intervals. The mechanism is depicted in Figure 3 and can be specified as follows:

1. Turn radio on.
2. Wait until $t = t_0 + t_{awake}$.
3. Turn radio off, but only if all communication has completed.
4. If communication has not completed, wait until it has completed or $t = t_0 + t_{awake} + t_{wait\_max}$.
5. Turn the radio off. Wait until $t = t_0 + t_{awake} + t_{sleep}$.
6. Repeat from step 1.

To implement this protocol in an event-driven model, we first need to identify a set of states around which the state machine can be designed. For this protocol, we can quickly identify three states: ON – the radio is on, WAITING – waiting for remaining communication to complete, and OFF – the radio is off. Figure 4 shows the resulting state machine, including the state transitions.

To implement this state machine, we use an explicit state variable, state, that can take on the values *ON*, *WAITING*, and *OFF*. We use an **if** statement to perform different actions depending on the value of the state variable. The code is

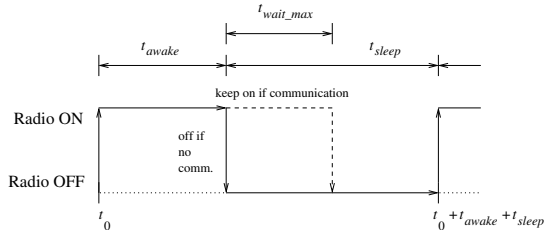

Figure 3. Hypothetical sensor network MAC protocol.

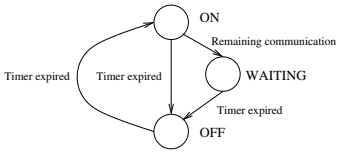

Figure 4. State machine realization of the radio sleep cycle of the example MAC protocol.

placed in an event handler function that is called whenever an event occurs. Possible events in this case are an expiration of a timer and the completion of communication. To simplify the code, we use two separate timers, timer and wait_timer, to keep track of the elapsed time. The resulting pseudocode is shown in Figure 1.

We note that this simple mechanism results in a fairly large amount of code. The code that controls the state machine constitutes more than one third of the total lines of code. Also, the six-step structure of the mechanism is not immediately evident from the code.

When implementing the radio sleep cycle mechanism with protothreads we can use the PT_WAIT_UNTIL() statement to wait for the timers to expire. Figure 2 shows the resulting pseudocode code. We see that the code is shorter than the event-driven version from Figure 1 and that the code more closely follows the specification of the mechanism.

2.4 Yielding Protothreads

Experience with rewriting event-driven state machines to protothreads revealed the importance of an unconditional blocking wait, PT_YIELD(). PT_YIELD() performs an single unconditional blocking wait that temporarily blocks the protothread until the next time the protothread is invoked. At the next invocation the protothread continues executing the code following the PT_YIELD() statement.

With the addition of the PT_YIELD() operation, protothreads are similar to stackless coroutines, much like cooperative multi-threading is similar to stackful coroutines.

2.5 Hierarchical Protothreads

While many programs can be readily expressed with a single protothread, more complex operations may need to be decomposed in a hierarchical fashion. Protothreads support this through an operation, PT_SPAWN(), that initializes a child protothread and blocks the current protothread until the child protothread has either ended with PT_END or exited with PT_EXIT. The child protothread is scheduled by the parent protothread; each time the parent protothread is invoked by the underlying system, the child protothread is invoked through the PT_SPAWN() statement. The memory for the state of the child protothread typically is allocated in a local variable of the parent protothread.

As a simple example of how hierarchical protothreads work, we consider a hypothetical data collection protocol that runs in two steps. The protocol first propagates data interest messages through the network. It then continues to propagate data messages back to where the interest messages came from. Both interest messages and data messages are transmitted in a reliable way: messages are retransmitted until an acknowledgment message is received.

```
reliable_send(message):
  rxtimer: timer
  PT_BEGIN
  do
    rxtimer ← t_retransmission
    send(message)
    PT_WAIT_UNTIL(ack_received() or expired(rxtimer))
  until (ack_received())
  PT_END

data_collection_protocol
  child_state: protothread_state
  PT_BEGIN
  while (running)
    while (interests_left_to_relay())
      PT_WAIT_UNTIL(interest_message_received())
      send_ack()
      PT_SPAWN(reliable_send(interest), child_state)
    while (data_left_to_relay())
      PT_WAIT_UNTIL(data_message_received())
      send_ack()
      PT_SPAWN(reliable_send(data), child_state)
  PT_END
```

Figure 5. Hypothetical data collection protocol implemented with hierarchical protothreads, in pseudocode.

Figure 5 shows this protocol implemented using hierarchical protothreads. The program consists of a main protothread, data_collection_protocol, that invokes a child protothread, reliable_send, to do transmission of the data.

2.6 Local Continuations

Local continuations are the low-level mechanism that underpins protothreads. When a protothread blocks, the state of the protothread is stored in a local continuation. A local continuation is similar to ordinary continuations [31] but, unlike a continuation, a local continuation does not capture the program stack. Rather, a local continuation only captures the state of execution inside a single function. The state of execution is defined by the continuation point in the function where the program is currently executing and the values of the function's local variables. The protothreads mechanism only requires that those variables that are actually used across a blocking wait to be stored. However, the current C-based prototype implementations of local continuations depart from this and do not store any local variables.

A local continuation has two operations: *set* and *resume*. When a local continuation is *set*, the state of execution is stored in the local continuation. This state can then later be restored with the *resume* operation. The state captured by a local continuation does not include the history of functions that have called the function in which the local continuation was *set*. That is, the local continuation does not contain the stack, but only the state of the current function.

A protothread consists of a function and a single local continuation. The protothread's local continuation is *set* before each PT_WAIT_UNTIL() statement. If the condition is false and the wait is to be performed, the protothread is suspended by returning control to the function that invoked the protothread's function. The next time the protothread func-

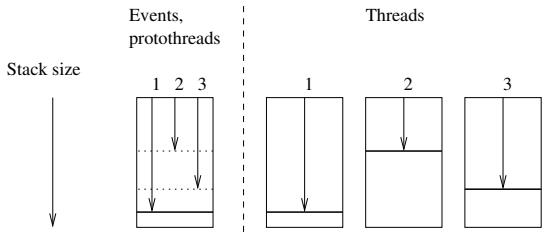

Figure 6. The stack memory requirements for three event handlers, the three event handlers rewritten with protothreads, and the equivalent functions running in three threads. Event handlers and protothreads run on the same stack, whereas each thread runs on a stack of its own.

tion is invoked, the protothread *resumes* the local continuation. This effectively causes the program to execute a jump to the conditional blocking wait statement. The condition is reevaluated and either blocks or continues its execution.

3 Memory Requirements

Programs written with an event-driven state machine need to store the state of the state machine in a variable in memory. The state can be stored in a single byte unless the state machine has more than 256 states. While the actual program typically stores additional state as program variables, the single byte needed for storing the explicit state constitutes the memory overhead of the state machine. The same program written with protothreads also needs to store the same program variables, and will therefore require exactly the same amount memory as the state machine implementation. The only additional memory overhead is the size of the continuation point. For the prototype C-based implementations, the size of the continuation point is two bytes on the MSP430 and three bytes for the AVR.

In a multi-threading system each thread requires its own stack. Typically, in memory-constrained systems this memory must be statically reserved for the thread and cannot be used for other purposes, even when the thread is not currently executing. Even for systems with dynamic stack memory allocation, thread stacks usually are over-provisioned because of the difficulties of predicting the maximum stack usage of a program, For example, the default stack size for one thread in the Mantis system [2] is 128 bytes, which is a large part of the memory in a system with a few kilobytes of RAM.

In contrast to multi-threading, for event-driven state machines and protothreads all programs run on the same stack. The minimum stack memory requirement is therefore the same as the maximum stack usage of all programs. The minimum memory requirement for stacks in a multi-threaded system, however, is the sum of the maximum stack usage of all threads. This is illustrated in Figure 6.

4 Replacing State Machines with Protothreads

We analyzed a number of existing event-driven programs and found that most control-flow state machines could be decomposed to three primitive patterns: sequences, iterations, and selections. While our findings hold for a number of memory-constrained sensor network and embedded programs, our findings are not new in general; Behren et al. [35] found similar results when examining several event-driven systems. Figure 7 shows the three primitives. In this section, we show how these state machine primitives map onto protothread constructs and how those can be used to replace state machines.

Figures 8 and 9 show how to implement the state machine patterns with protothreads. Protothreads allow the programmer to make use of the control structures provided by the programming language: the selection and iteration patterns map onto **if** and **while** statements.

To rewrite an event-driven state machine with protothreads, we first analyse the program to find its state machine. We then map the state machine patterns from Figure 7 onto the state machine from the event-driven program. When the state machine patterns have been identified, the program can be rewritten using the code patterns in Figures 8 and 9.

As an illustration, Figure 10 shows the state machine from the radio sleep cycle of the example MAC protocol in Section 2.3, with the iteration and sequence state machine patterns identified. From this analysis the protothreads-based code in Figure 2 can be written.

5 Implementation

We have developed two prototype implementations of protothreads that use only the C preprocessor. The fact that the implementations only depend on the C preprocessor adds the benefit of full portability across all C compilers and of not requiring extra tools in the compilation tool chain. However, the implementations depart from the protothreads mechanism in two important ways: automatic local variables are not saved across a blocking wait statement and C switch and case statements cannot be freely intermixed with protothread-based code. These problems can be solved by implementing protothreads as a special precompiler or by integrating protothreads into existing preprocessor-based languages and C language extensions such as nesC [15].

5.1 Prototype C Preprocessor Implementations

In the prototype C preprocessor implementation of protothreads the protothread statements are implemented as C preprocessor macros that are shown in Figure 11. The protothread operations are a very thin layer of code on top of the local continuation mechanism. The *set* and *resume* operations of the local continuation are implemented as an LC_SET() and the an LC_RESUME() macro. The prototype implementations of LC_SET() and LC_RESUME() depart from the mechanism specified in Section 2.6 in that auto-

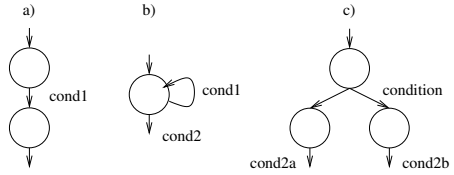

Figure 7. Two three primitive state machines: a) sequence, b) iteration, c) selection.

```
a_sequence:
  PT_BEGIN
  (* ... *)
  PT_WAIT_UNTIL(cond1)
  (* ... *)
  PT_END
```

```
an_iteration:
  PT_BEGIN
  (* ... *)
  while (cond1)
    PT_WAIT_UNTIL(cond1 or
                  cond2)
  (* ... *)
  PT_END
```

Figure 8. Pseudocode implementation of the sequence and iteration patterns with protothreads.

```
a_selection:
  PT_BEGIN
  (* ... *)
  if (condition)
    PT_WAIT_UNTIL(cond2a)
  else
    PT_WAIT_UNTIL(cond2b)
  (* ... *)
  PT_END
```

Figure 9. Pseudocode implementation of the selection pattern with a protothread.

matic variables are not saved, but only the continuation point of the function.

The PT_BEGIN() statement, which marks the start of a protothread, is implemented with a single LC_RESUME() statement. When a protothread function is invoked, the LC_RESUME() statement will resume the local continuation stored in the protothread's state structure, thus performing an unconditional jump to the last place where the local continuation was set. The resume operation will not perform the jump the first time the protothread function is invoked.

The PT_WAIT_UNTIL() statement is implemented with a LC_SET() operation followed by an `if` statement that performs an explicit `return` if the conditional statement evaluates to false. The returned value lets the caller know that the protothread blocked on a PT_WAIT_UNTIL() statement. PT_END() and PT_EXIT() immediately return to the caller.

To implement yielding protothreads, we need to change the implementation of PT_BEGIN() and PT_END() in addition to implementing PT_YIELD(). The implementation of PT_YIELD() needs to test whether the protothread has yielded or not. If the protothread has yielded once, then the protothread should continue executing after the PT_YIELD() statement. If the protothread has not yet yielded, it should perform a blocking wait. To implement this, we add an automatic variable, which we call `yielded` for the purpose of this discussion, to the protothread. The `yielded` variable is initialized to one in the PT_BEGIN() statement. This ensures that the variable will be initialized every time the protothread is invoked. In the implementation of PT_YIELD(), we set the variable to zero, and perform a PT_WAIT_UNTIL() that blocks until the variable is nonzero. The next time the protothread is invoked, the conditional statement in the PT_WAIT_UNTIL() is reevaluated. Since the `yielded` variable now has been reinitialized to one, the PT_WAIT_UNTIL() statement will not block. Figure 12 shows this implementation of PT_YIELD() and the updated PT_BEGIN() and PT_END() statements.

The implementation of PT_SPAWN(), which is used to implement hierarchical protothreads, is shown in Figure 13. It initializes the child protothread and invokes it every time the current protothread is invoked. The PT_WAIT_UNTIL() blocks until the child protothread has exited or ended.

We now discuss how the local continuation functions LC_SET() and LC_RESUME() are implemented.

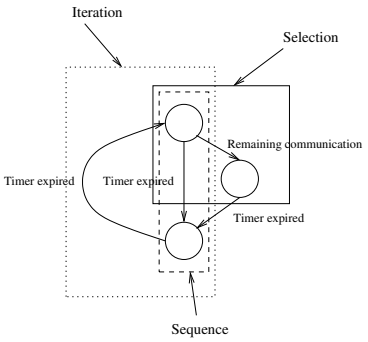

Figure 10. The state machine from the example radio sleep cycle mechanism with the iteration and sequence patterns identified.

```
struct pt { lc_t lc };
#define PT_WAITING 0
#define PT_EXITED  1
#define PT_ENDED   2
#define PT_INIT(pt)           LC_INIT(pt->lc)
#define PT_BEGIN(pt)          LC_RESUME(pt->lc)
#define PT_END(pt)            LC_END(pt->lc);     \
                              return PT_ENDED
#define PT_WAIT_UNTIL(pt, c) LC_SET(pt->lc);     \
                              if(!(c))            \
                                return PT_WAITING
#define PT_EXIT(pt)           return PT_EXITED
```

Figure 11. C preprocessor implementation of the main protothread operations.

```
#define PT_BEGIN(pt) { int yielded = 1;          \
                       LC_RESUME(pt->lc)
#define PT_YIELD(pt)   yielded = 0;              \
                       PT_WAIT_UNTIL(pt, yielded)
#define PT_END(pt)     LC_END(pt->lc);           \
                       return PT_ENDED; }
```

Figure 12. Implementation of the PT_YIELD() operation and the updated PT_BEGIN() and PT_END() statements.

```
#define PT_SPAWN(pt, child, thread)              \
        PT_INIT(child);                          \
        PT_WAIT_UNTIL(pt, thread != PT_WAITING)
```

Figure 13. Implementation of the PT_SPAWN() operation

```
typedef void * lc_t;
#define LC_INIT(c)   c = NULL
#define LC_RESUME(c) if(c) goto *c
#define LC_SET(c)    { __label__ r; r: c = &&r; }
#define LC_END(c)
```

Figure 14. Local continuations implemented with the GCC labels-as-values C extension.

```
typedef unsigned short lc_t;
#define LC_INIT(c)   c = 0
#define LC_RESUME(c) switch(c) { case 0:
#define LC_SET(c)    c = __LINE__; case __LINE__:
#define LC_END(c)    }
```

Figure 15. Local continuations implemented with the C switch statement.

```
 1 int sender(pt) {
 2   PT_BEGIN(pt);
 3
 4   /* ... */
 5   do {
 6
 7     PT_WAIT_UNTIL(pt,
 8                   cond1);
 9
10   } while(cond);
11   /* ... */
12   PT_END(pt);
13
14 }
```

```
int sender(pt) {
  switch(pt->lc) {
  case 0:
    /* ... */
    do {
      pt->lc = 8;
  case 8:
      if(!cond1)
      return PT_WAITING;
    } while(cond);
    /* ... */
  }
  return PT_ENDED;
}
```

Figure 16. Expanded C code with local continuations implemented with the C switch statement.

5.1.1 GCC C Language Extensions

The widely used GCC C compiler provides a special C language extension that makes the implementation of the local continuation operations straightforward. The C extension, called labels-as-values, makes it possible to save the address of a C label in a pointer. The C goto statement can then be used to jump to the previously captured label. This use of the goto operation is very similar to the unconditional jump most machine code instruction sets provide.

With the labels-as-values C extension, a local continuation simply is a pointer. The *set* operation takes the address of the code executing the operation by creating a C label and capturing its address. The *resume* operation resumes the local continuation with the C goto statement, but only if the local continuation previously has been *set*. The implementation of local continuations with C macros and the labels-as-values C language extension is shown in Figure 14. The LC_SET() operation uses the GCC __label__ extension to declare a C label that is local in scope. It then defines the label and stores the address of the label in the local continuation by using the GCC double-ampersand extension.

5.1.2 C Switch Statement

The main problem with the GCC C extension-based implementation of local continuations is that it only works with a single C compiler: GCC. We next show an implementation using only standard ANSI C constructs which uses the C switch statement in a non-obvious way.

Figure 15 shows local continuations implemented using the C switch statement. LC_RESUME() is an open switch statement, with a `case 0:` immediately following it. The `case 0:` makes sure that the code after the LC_RESUME() statement is always executed when the local continuation has been initialized with LC_INIT(). The implementation of LC_SET() uses the standard __LINE__ macro. This macro expands to the line number in the source code at which the LC_SET() macro is used. The line number is used as a unique identifier for each LC_SET() statement. The implementation of LC_END() is a single right curly bracket that closes the switch statement opened by LC_RESUME().

To better illustrate how the C switch-based implementation works, Figure 16 shows how a short protothreads-based program is expanded by the C preprocessor. We see that the resulting code is fairly similar to how the explicit state machine was implemented in Figure 1. However, when looking closer at the expanded C code, we see that the `case 8:` statement on line 7 appears inside the do-while loop, even though the switch statement appears outside of the do-while loop. This does seem surprising at first, but is in fact valid ANSI C code. This use of the switch statement is likely to first have been publicly described by Duff as part of Duff's Device [8]. The same technique has later been used by Tatham to implement coroutines in C [33].

5.2 Memory Overhead

The memory required for storing the state of a protothread, implemented either with the GCC C extension or the C switch statement, is two bytes; the C switch statement-based implementation requires two bytes to store the 16-bit line number identifier of the local continuation. The C extension-based implementation needs to store a pointer to the address of the local continuation. The size of a pointer is processor-dependent but on the MSP430 a pointer is 16 bits, resulting in a two byte memory overhead. A pointer on the AVR is 24 bits, resulting in three bytes of memory overhead. However, the memory overhead is an artifact of the prototype implementations; a precompiler-based implementation would reduce the overhead to one byte.

5.3 Limitations of the Prototype Implementations

The two implementations of the local continuation mechanism described above introduce the limitation that automatic variables are not saved across a blocking wait. The C switch-based implementation also limits the use of the C switch statement together with protothread statements.

5.3.1 Automatic Variables

In the C-based prototype implementations, automatic variables—variables with function-local scope that are automatically allocated on the stack—are not saved in the local continuation across a blocking wait. While automatic variables can still be used inside a protothread, the contents of the variables must be explicitly saved before executing a wait statement. Many C compilers, including GCC, detect if auto-

matic local variables are used across a blocking protothreads statement and issues a warning message.

While automatic variables are not preserved across a blocking wait, static local variables are preserved. Static local variables are variables that are local in scope but allocated in the data section of the memory rather than on the stack. Since static local variables are not placed on the stack, they are not affected by the use of blocking protothreads statements. For functions that do not need to be reentrant, static local variables allow the programmer to use local variables inside the protothread.

For reentrant protothreads, the limitation on the use of automatic variables can be handled by using an explicit state object, much in the same way as is commonly done in purely event-driven programs. It is, however, the responsibility of the programmer to allocate and maintain such a state object.

5.3.2 Constraints on Switch Constructs

The implementation of protothreads using the C switch statements imposes a restriction on programs using protothreads: programs cannot utilize switch statements together with protothreads. If a switch statement is used by the program using protothreads, the C compiler will in some cases emit an error, but in most cases the error is not detected by the compiler. This is troublesome as it may lead to unexpected run-time behavior which is hard to trace back to an erroneous mixture of one particular implementation of protothreads and switch statements. We have not yet found a suitable solution for this problem other than using the GCC C extension-based implementation of protothreads.

5.3.3 Possible C Compiler Problems

It could be argued that the use of a non-obvious, though standards-compliant, C construct can cause problems with the C compiler because the nested switch statement may not be properly tested. We have, however, tested protothreads on a wide range of C compilers and have only found one compiler that was not able to correctly parse the nested C construct. In this case, we contacted the vendor who was already aware of the problem and immediately sent us an updated version of the compiler. We have also been in touch with other C compiler vendors, who have all assured us that protothreads work with their product.

5.4 Alternative Approaches

In addition to the implementation techniques described above, we examine two alternative implementation approaches: implementation with assembly language and with the C language functions setjmp and longjmp.

5.4.1 Assembly Language

We have found that for some combinations of processors and C compilers it is possible to implement protothreads and local continuations by using assembly language. The *set* of the local continuations is then implemented as a C function that captures the return address from the stack and stores it in the local continuation, along with any callee save registers. Conversely, the *resume* operation would restore the saved registers from the local continuation and perform an unconditional jump to the address stored in the local continuation. The obvious problem with this approach is that it requires a porting effort for every new processor and C compiler. Also, since both a return address and a set of registers need to be stored in the local continuation, its size grows. However, we found that the largest problem with this approach is that some C compiler optimizations will make the implementation difficult. For example, we were not able to produce a working implementation with this method for the Microsoft Visual C++ compiler.

5.4.2 With C setjmp and longjmp Functions

While it at first seems possible to implement the local continuation operations with the setjmp and longjmp functions from the standard C library, we have seen that such an implementation causes subtle problems. The problem is because the setjmp and longjmp function store and restore the stack pointer, and not only the program counter. This causes problems when the protothread is invoked through different call paths since the stack pointer is different with different call paths. The *resume* operation would not correctly resume a local continuation that was *set* from a different call path.

We first noticed this when using protothreads with the uIP TCP/IP stack. In uIP application protothreads are invoked from different places in the TCP/IP code depending on whether or not a TCP retransmission is to take place.

5.4.3 Stackful Approaches

By letting each protothread run on its own stack it would be possible to implement the full protothread mechanism, including storage of automatic variables across a blocking wait. With such an implementation the stack would be switched to the protothread's own stack by the PT_BEGIN operation and switched back when the protothread blocks or exits. This approach could be implemented with a coroutine library or the multi-threading library of Contiki. However, this implementation would result in a memory overhead similar to that of multi-threading because each invocation of a protothread would require the same amount of stack memory as the equivalent protothread running in a thread of its own due to the stack space required by functions called from within the protothread.

Finally, a promising alternative method is to store a copy the stack frame of the protothread function in the local continuation when the protothread blocks. This saves all automatic variables of the protothread function across a blocking wait, including variables that are not used after the blocking wait. Since all automatic variables are saved, this approach have a higher memory overhead. Furthermore, this approach requires both C compiler-specific and CPU architecture-specific code, thus reducing the portability of the implementation. However, the extra porting effort may be outweighed by the benefits of storing automatic variables across blocking waits. We will continue to pursue this as future work.

6 Evaluation

To evaluate protothreads we first measure the reduction in code complexity that protothreads provide by reimplementing a set of event-driven programs with protothreads and measure the complexity of the resulting code. Second, we measure the memory overhead of protothreads compared to the memory overhead of an event-driven state machine. Third, we compare the execution time overhead of protothreads with that of event-driven state machines.

6.1 Code Complexity Reduction

To measure the code complexity reduction of protothreads we reimplement parts of a number of event-driven applications with protothreads: XNP [20], the previous default over-the-air programming program from TinyOS; the buffer management module of TinyDB [27], a database engine for TinyOS; radio protocol drivers for the Chipcon CC1000 and RF Monolithics TR1001 radio chips; the SMTP client in the uIP embedded TCP/IP stack and a code propagation program from the Contiki operating system. The state machines in XNP, TinyDB, and the CC1000 drivers were rewritten by applying the method for replacing state machines with protothreads from Section 4 whereas the TR1001 driver, the uIP SMTP client and the Contiki code propagation were rewritten from scratch.

We use three metrics to measure the complexity of the programs we reimplemented with protothreads: the number of explicit states, the number of explicit state transitions, as well as the lines of code of the reimplemented functions.

All reimplemented programs consist of complex state machines. Using protothreads, we were able to entirely remove the explicit state machines for most programs. For all programs, protothreads significantly reduce the number of state transitions and lines of code.

The reimplemented programs have undergone varying amounts of testing. The Contiki code propagation, the TR1001 low-level radio driver, and the uIP SMTP client are well tested and are currently used on a daily basis in live systems, XNP and TinyDB have been verified to be working but not heavily tested, and the CC1000 drivers have been tested and run in simulation.

Furthermore, we have anecdotal evidence to support our hypothesis that protothreads are an alternative to state machines for embedded software development. The protothreads implementations have for some time been available as open source on our web page [9]. We know that at least ten embedded systems developers have successfully used protothreads to replace state machines for embedded software development. Also, our protothreads code have twice been recommended by experienced embedded developers in Jack Ganssle's embedded development newsletter [14].

6.1.1 XNP

XNP [20] is one of the in-network programming protocols used in TinyOS [19]. XNP downloads a new system image to a sensor node and writes the system image to the flash memory of the device. XNP is implemented on top of the event-driven TinyOS. Therefore, any operations in XNP that would be blocking in a threaded system have to be implemented as state machines. We chose XNP because it is a relatively complex program implemented on top of an event-driven system. The implementation of XNP has previously been analyzed by Jeong [20], which assisted us in our analysis. The implementation of XNP consists of a large switch statement with 25 explicit states, encoded as defined constants, and 20 state transitions. To analyze the code, we identified the state transitions from manual inspection of the code inside the switch statement.

Since the XNP state machine is implemented as one large switch statement, we expected it to be a single, complex state machine. But, when drawing the state machine from analysis of the code, it turned out that the switch statement in fact implements five different state machines. The entry points of the state machines are not immediately evident from the code, as the state of the state machine was changed in several places throughout the code.

The state machines we found during the analysis of the XNP program are shown in Figure 17. For reasons of presentation, the figure does not show the IDLE and ACK states. Almost all states have transitions to one of these states. If an XNP operation completes successfully, the state machine goes into the ACK state to transmit an acknowledgment over the network. The IDLE state is entered if an operation ends with an error, and when the acknowledgment from the ACK state has been transmitted.

In the figure we clearly see many of the state machine patterns from Figure 7. In particular, the sequence pattern is evident in all state machines. By using the techniques described in Section 4 we were able to rewrite all state machines into protothreads. Each state machine was implemented as its own protothread.

The IDLE and ACK states are handled in a hierarchical protothread. A separate protothread is created for sending the acknowledgment signal. This protothread is spawned from the main protothread every time the program logic dictates that an acknowledgment should be sent.

6.1.2 TinyDB

TinyDB [27] is a small database engine for the TinyOS system. With TinyDB, a user can query a wireless sensor network with a database query language similar to SQL. TinyDB is one of the largest TinyOS programs available.

In TinyOS long-latency operations are split-phase [15]. Split-phase operations consist of two parts: a request and a completion event. The request completes immediately, and the completion event is posted when the operation has completed. TinyDB contains a large number of split-phase operations. Since programs written for TinyOS cannot perform a blocking wait, many complex operations in TinyDB are encoded as state machines.

To the state machines in TinyDB we analyze the TinyDB buffer management module, DBBufferC. DBBufferC uses the MemAlloc module to allocate memory. Memory allocation requests are performed from inside a function that drives the state machine. However, when the request is completed, the allocComplete event is handled by a different function. This event handler must handle the event different depending on the state of the state machine. In fact, the event handler itself implements a small piece of the entire state machine. The fact that the implementation of the state machine is distributed across different functions makes the analysis of the state machine difficult.

From inspection of the DBBufferC code we found the three state machines in Figure 18. We also found that there are more state machines in the code, but we were not able to adequately trace them because the state transitions were scattered around the code. By rewriting the discovered state machines with protothreads, we were able to completely remove the explicit state machines.

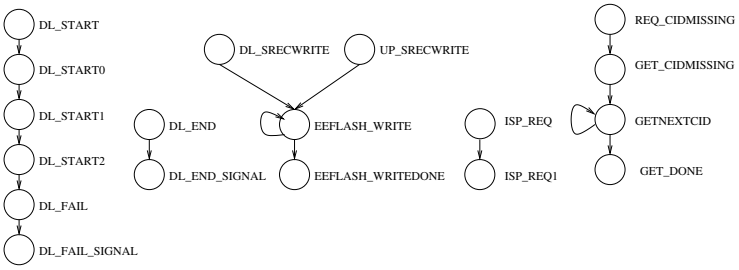

Figure 17. XNP state machines. The names of the states are from the code. The IDLE and ACK states are not shown.

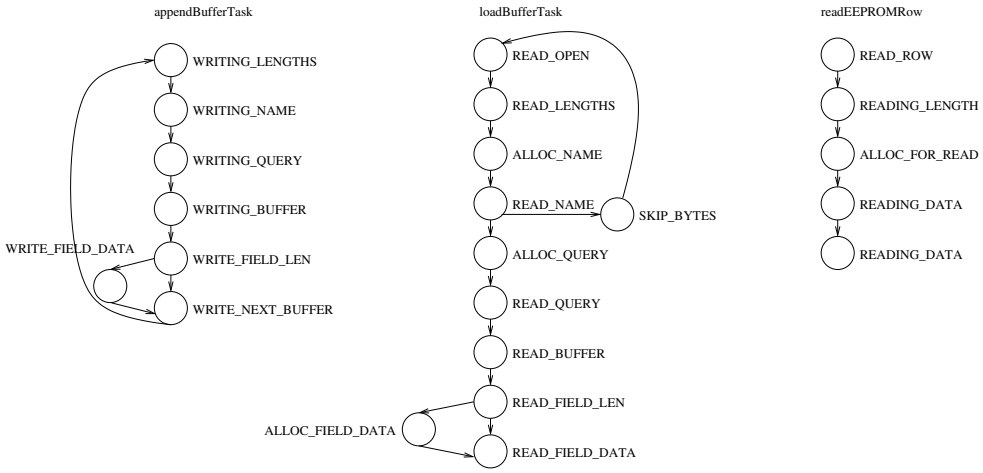

Figure 18. Three state machines from TinyDB.

6.1.3 Low Level Radio Protocol Drivers

The Chipcon CC1000 and RF Monolithics TR1001 radio chips are used in many wireless sensor network devices. Both chips provide a very low-level interface to the radio. The chips do not perform any protocol processing themselves but interrupt the CPU for every incoming byte. All protocol functionality, such as packet framing, header parsing, and MAC protocol must be implemented in software.

We analyze and rewrite CC1000 drivers from the Mantis OS [2] and from SOS [17], as well as the TR1001 driver from Contiki [12]. All drivers are implemented as explicit state machines. The state machines run in the interrupt handlers of the radio interrupts.

The CC1000 driver in Mantis has two explicit state machines: one for handling and parsing incoming bytes and one for handling outgoing bytes. In contrast, both the SOS CC1000 driver and the Contiki TR1001 drivers have only one state machine that parses incoming bytes. The state machine that handles transmissions in the SOS CC1000 driver is shown in Figure 19. The structures of the SOS CC1000 driver and the Contiki TR1001 driver are very similar.

With protothreads we could replace most parts of the state machines. However, for both the SOS CC1000 driver and the Contiki TR1001 drivers, we kept a top-level state machine. The reason for this is that those state machines were not used to implement control flow. The top-level state machine in the SOS CC1000 driver controlled if the driver was currently transmitting or receiving a packet, or if it was finding a synchronization byte.

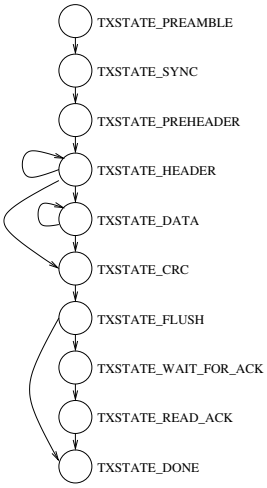

Figure 19. Transmission state machine from the SOS CC1000 driver.

6.1.4 uIP TCP/IP Stack

The uIP TCP/IP stack [10] is designed for memory-constrained embedded systems and therefore has a very low memory overhead. It is used in embedded devices from well over 30 companies, with applications ranging from pico-satellites to car traffic monitoring systems. To reduce the memory overhead uIP follows the event-driven model. Application programs are implemented as event handlers and

Program	Code size, before (bytes)	Code size, after (bytes)	Increase
XNP	931	1051	13%
TinyDB DBBufferC	2361	2663	13%
Mantis CC1000	994	1170	18%
SOS CC1000	1912	2165	13%
Contiki TR1001	823	836	2%
uIP SMTP	1106	1901	72%
Contiki code prop.	1848	1426	-23%

Table 2. Code size before and after rewriting with protothreads.

to achieve blocking waits, application programs need to be written as explicit state machines. We have rewritten the SMTP client in uIP with protothreads and were able to completely remove the state machines.

6.1.5 Contiki

The Contiki operating system [12] for wireless sensor networks is based on an event-driven kernel, on top of which protothreads provide a thread-like programming style. The first version of Contiki was developed before we introduced protothreads. After developing protothreads, we found that they reduced the complexity of writing software for Contiki.

For the purpose of this paper, we measure the implementation of a distribution program for distributing and receiving binary code modules for the Contiki dynamic loader [11]. The program was initially implemented without protothreads but was later rewritten when protothreads were introduced to Contiki. The program can be in one of three modes: (1) receiving a binary module from a TCP connection and loads it into the system, (2) broadcasting the binary module over the wireless network, and (3) receiving broadcasts of a binary module from a nearby node and loading it into memory.

When rewriting the program with protothreads, we removed most of the explicit state machines, but kept four states. These states keep track in which mode the program is: if it is receiving or broadcasting a binary module.

6.1.6 Results

The results of reimplementing the programs with protothreads are presented in Table 1. The lines of code reported in the table are those of the rewritten functions only. We see that in all cases the number of states, state transitions, and lines of code were reduced by rewriting the programs with protothreads. In most cases the rewrite completely removed the state machine. The total average reduction in lines of code is 31%. For the programs rewritten by applying the replacement method from Section 4 (XNP, TinyDB, and the CC1000 drivers) the average reduction is 23% and for the programs that were rewritten from scratch (the TR1001 driver, the uIP SMTP client, and the Contiki code propagation program) the average reduction is 41%.

Table 2 shows the compiled code size of the rewritten functions when written as a state machine and with protothreads. We see that the code size increases in most cases, except for the Contiki code propagation program. The average increase for the programs where the state machines were replaced with protothreads by applying the method from Section 4 is 14%. The Contiki TR1001 driver is only marginally larger when written with protothreads. The uIP SMTP client, on the other hand, is significantly larger when written with protothreads rather than with a state machine. The reason for this is that the code for creating SMTP message strings could be optimized through code reuse in the state machine-based implementation, something which was not possible in the protothreads-based implementation without significantly sacrificing readability. In contrast with the uIP SMTP client, the Contiki code propagation program is significantly smaller when written with protothreads. Here, it was possible to optimize the protothreads-based code by code reuse, which was not readily possible in the state machine-based implementation.

	State machine	Protothread	Thread
Contiki TR1001 driver	1	2	18
Contiki code propagation	1	2	34

Table 3. Memory overhead in bytes for the Contiki TR1001 driver and the Contiki code propagation on the MSP430, implemented with a state machine, a protothread, and a thread.

We conclude that protothreads reduce the number of states and state transitions and the lines of code, at the price of an increase in code size. The size of the increase, however, depends on the properties of the particular program rewritten with protothreads. It is therefore not possible to draw any general conclusions from measured increase in code size.

6.2 Memory Overhead

We measured the stack size required for the Contiki TR1001 driver and the Contiki code propagation mechanism running on the MSP430 microcontroller. We measure the stack usage by filling the stack with a known byte pattern, running the TR1001 driver, and inspecting the stack to see how much of the byte pattern that was overwritten.

Neither the Contiki code propagation mechanism nor the Contiki TR1001 driver use the stack for keeping state in program variables. Therefore the entire stack usage of the two programs when running as threads is overhead.

We compare the memory overhead of the Contiki TR1001 driver and the Contiki code propagation running as threads with the memory overhead of a state machine and protothreads in Table 3.

6.3 Run-time Overhead

To evaluate the run-time overhead of protothreads we counted the machine code instruction overhead of protothreads, compared to a state machine, for the MSP430 and the AVR microcontrollers. Furthermore, we measured the execution time for the driver for the TR1001 radio chip from Contiki, implemented both as a state machine and as a protothread. We also compare the numbers with measurements on a cooperative multi-threading implementation of said program. We used the cooperative user space multi-threading library from Contiki [12] which is a standard stack-switching multi-threading library.

Program	States, before	States, after	Transitions, before	Transitions, after	Lines of code, before	Lines of code, after	Reduction, percentage
XNP	25	-	20	-	222	152	32%
TinyDB	23	-	24	-	374	285	24%
Mantis CC1000 driver	15	-	19	-	164	127	23%
SOS CC1000 driver	26	9	32	14	413	348	16%
Contiki TR1001 driver	12	3	32	3	152	77	49%
uIP SMTP client	10	-	10	-	223	122	45%
Contiki code propagation	6	4	11	3	204	144	29%

Table 1. The number of explicit states, explicit state transitions, and lines of code before and after rewriting with protothreads.

	State machine	Proto-thread	Yielding protothread
MSP430	9	12	17
AVR	23	34	45

Table 4. Machine code instructions overhead for a state machine, a protothread, and a yielding protothread.

6.3.1 Machine Code Instruction Overhead

To analyze the number of additional machine code instructions in the execution path for protothreads compared to state machines, we compiled our program using the GCC C compiler version 3.2.3 for the MSP430 microcontroller and GCC version 3.4.3 for the AVR microcontroller.

With manual inspection of the generated object machine code we counted the number of machine code instructions needed for the switch and case statements in a state machine function and for the protothread operations in a protothread function. The results are given in Table 4. The absolute overhead for protothreads over the state machine is very small: three machine code instructions for the MSP430 and 11 for the AVR. In comparison, the number of instructions required to perform a context switch in the Contiki implementation of cooperative multi-threading is 51 for the MSP430 and 80 for the AVR.

The additional instructions for the protothread in Table 4 are caused by the extra case statement that is included in the implementation of the PT_BEGIN operation.

6.3.2 Execution Time Overhead

To measure the execution time overhead of protothreads over that of an event-driven state machine, we implemented the low-level radio protocol driver for the RFM TR1001 radio chip from Contiki using both an event-driven state machine and a yielding protothread. We measured the execution time by feeding the driver with data from 1000 valid radio packets and measured the average execution time of the driver's function. The time was measured using a periodic timer interrupt driven by the MSP430 timer A1 at a rate of 1000 Hz. We set the clock speed of the MSP430 digitally controlled oscillator to 2.4576 MHz. For the protothreads-based implementation we measured both the GCC C extension-based and the C switch statement-based implementations of the local continuations operations.

The measurement results are presented in Table 5. We see that the average execution time overhead of protothreads is low: only about five cycles per invocation for the GCC C extension-based implementation and just over ten cycles per invocation for the C switch-based implementation. The results are consistent with the machine code overhead in Table 4. We also measured the execution time of the radio driver rewritten with cooperative multi-threading and found it to be approximately three times larger than that of the protothread-based implementation because of the overhead of the stack switching code in the multi-threading library.

Because of the low execution time overhead of protothreads we conclude that protothreads are usable even for interrupt handlers with tight execution time constraints.

7 Discussion

We have been using the prototype implementations of protothreads described in this paper in Contiki for two years and have found that the biggest problem with the prototype implementations is that automatic variables are not preserved across a blocking wait. Our workaround is to use static local variables rather than automatic variables inside protothreads. While the use of static local variables may be problematic in the general case, we have found it to work well for Contiki because of the small scale of most Contiki programs. Also, as many Contiki programs do not need to be reentrant, the use of static local variables work well.

Code organization is different for programs written with state machines and with protothreads. State machine-based programs tend to consist either of a single large function containing a large state machine or of many small functions where the state machine is difficult to find. On the contrary, protothreads-based programs tend to be based around a single protothread function that contains the high-level logic of the program. If the underlying event system calls different functions for every incoming event, a protothreads-based program typically consists of a single protothread function and a number of small event handlers that invoke the protothread when an event occurs.

8 Related Work

Research in the area of software development for sensor networks has led to a number of new abstractions that aim at simplifying sensor network programming [1, 4]. Approaches with the same goal include virtual machines [25] and macro-programming of sensors [16, 29, 37]. Protothreads differ from these sensor network programming abstractions in that we target the difficulty of low-level event-driven programming rather than the difficulty of developing application software for sensor networks.

Compiler optimization	State machine (ms)	Protothreads, GCC C extension (ms)	Protothreads, C switch statement (ms)	State machine (cycles)	Protothreads, GCC C extension (cycles)	Protothreads, C switch statement (cycles)
Size (-Os)	0.0373	0.0397	0.0434	91.67	97.56	106.7
Speed (-O1)	0.0369	0.0383	0.0415	90.69	94.12	102.0

Table 5. Mean execution time in milliseconds and processor cycles for a single invocation of the TR1001 input driver under Contiki on the MSP430 platform.

Kasten and Römer [21] have also identified the need for new abstractions for managing the complexity of event-triggered state machine programming. They introduce OSM, a state machine programming model based on Harel's StateCharts[18] and use the Esterel language. The model reduces both the complexity of the implementations and the memory usage. Their work is different from protothreads in that they help programmers manage their state machines, whereas protothreads are designed to reduce the number of state machines. Furthermore, OSM requires support from an external OSM compiler to produce the resulting C code, whereas the prototype implementations of protothreads only make use of the regular C preprocessor.

Simpson [32] describes a cooperative mini-kernel mechanism for C++ which is very similar to our protothreads in that it was designed to replace state machines. Simpson's mechanism also uses a single stack. However, it needs to implement its own stack to track the location of a yield point. In contrast, protothreads do not use a stack, but hold all their state in a 16-bit integer value.

Lauer and Needham [24] proved, essentially, that events and threads are duals of each other and that the same program can be written for either of the two systems. With protothreads, we put this into practice by actually rewriting event-driven programs with blocking wait semantics.

Protothreads are similar to coroutines [22] in the sense that a protothread continues to execute at the point of the last return from the function. In particular, protothreads are similar to asymmetric coroutines [7], just as cooperative multi-threading is similar to asymmetric coroutines. However, unlike coroutines and cooperative multi-threading, protothreads are stackless and can only block at the top level of the protothread function.

Dabek et al. [6] present libasynch, a C++ library that assists the programmer in writing event-driven programs. The library provides garbage collection of allocated memory as well as a type-safe way to pass state between callback functions. However, the libasynch library does not provide sequential execution and software written with the library cannot use language statements to control program flow across blocking calls.

The Capriccio system by von Behren et al. [36] shows that in a memory-rich environment, threads can be made as memory efficient as events. This requires modification to the C compiler so that it performs dynamic memory allocation of stack space during run-time. However, as dynamic memory allocation quickly causes memory fragmentation in the memory-constrained devices for which the protothreads mechanism is designed, dynamic allocation of stack memory is not feasible.

Cunningham and Kohler [5] develop a library to assist programmers of event-driven systems in program analysis, along with a tool for visualizing callback chains, as well as a tool for verifying properties of programs implemented using the library. Their work is different from ours in that they help the programmer manage the state machines for event-driven programs, whereas protothreads are designed to replace such state machines.

Adya et al. [3] discuss the respective merits of event-driven and threaded programming and present a hybrid approach that shows that the event-driven and multi-threaded code can coexist in a unified concurrency model. The authors have developed a set of adaptor functions that allows event-driven code to call threaded code, and threaded code to call event-driven code, without requiring that the caller has knowledge about the callee's approach.

State machines are a powerful tool for developing real-time systems. Abstractions such as Harel's StateCharts [18] are designed to help developers to develop and manage state machines and are very valuable for systems that are designed as state machines. Protothreads, in contrast, are intended to replace state machines with sequential C code. Also, protothreads do not provide any mechanisms for assisting development of hard real-time systems.

9 Conclusions

We present protothreads, a novel abstraction for memory-constrained embedded systems. Due to memory-constraints, such systems are often based on an event-driven model. Experience has shown that event-driven programming is difficult because the lack of a blocking wait abstraction forces programmers to implement control flow with state machines.

Protothreads simplify programming by providing a conditional blocking wait operation, thereby reducing the need for explicit state machines. Protothreads are inexpensive: the memory overhead is only two bytes per protothread.

We develop two prototype protothreads implementations using only C preprocessor and evaluate the usefulness of protothreads by reimplementing some widely used event-driven programs using protothreads. Our results show that for most programs the explicit state machines could be entirely removed. Furthermore, protothreads significantly reduce the number of state transitions and lines of code. The execution time overhead of protothreads is on the order of a few processor cycles. We find the code size of a program written with protothreads to be slightly larger than the equivalent program written as a state machine.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European

Commission under contract IST-004536-RUNES. Thanks go to Kay Römer and Umar Saif for reading and suggesting improvements on drafts of this paper, and to our paper shepherd Philip Levis for his many insightful comments that significantly helped to improve the paper.

10 References

[1] T. Abdelzaher, J. Stankovic, S. Son, B. Blum, T. He, A. Wood, and C. Lu. A communication architecture and programming abstractions for real-time embedded sensor networks. In *Workshop on Data Distribution for Real-Time Systems*, Providence, RI, USA, May 2003.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 50–59, 2003.

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[4] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A programming model for event-driven embedded systems. In *Proc. of the 18th Annual ACM Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA, March 2003.

[5] R. Cunningham and E. Kohler. Making events less slippery with eel. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)*, Santa Fee, New Mexico, June 2005. IEEE Computer Society.

[6] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 2002 SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[7] A. L. de Moura and R. Ierusalimschy. Revisiting coroutines. MCC 15/04, PUC-Rio, Rio de Janeiro, RJ, June 2004.

[8] T. Duff. Unwinding loops. Usenet news article, net.lang.c, Message-ID: <2748@alice.UUCP>, May 1984.

[9] A. Dunkels. Protothreads web site. Web page. Visited 2006-04-06. http://www.sics.se/~adam/pt/

[10] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, May 2003.

[11] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006*, Boulder, Colorado, USA, 2006.

[12] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[13] A. Dunkels, O. Schmidt, and T. Voigt. Using protothreads for sensor node programming. In *Proc. of the Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, Stockholm, Sweden, June 2005.

[14] J. Ganssle. The embedded muse. Monthly newsletter. http://www.ganssle.com/tem-back.htm

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[16] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *Proc. of Distributed Computing in Sensor Systems (DCOSS)'05*, Marina del Rey, CA, USA, June 2005.

[17] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.

[18] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[19] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[20] J. Jeong. Analysis of xnp network reprogramming module. Web page, October 2003. Visited 2006-04-06. http://www.cs.berkeley.edu/~jaein/cs294_1/xnp_anal.htm

[21] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, Los Angeles, USA, April 2005.

[22] D. E. Knuth. *The art of computer programming, volume 1: fundamental algorithms (2nd edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[23] Framework Labs. Protothreads for Objective-C/Cocoa. Visited 2006-04-06. http://www.frameworklabs.de/protothreads.html

[24] H. C. Lauer and R. M. Needham. On the duality of operating systems structures. In *Proc. Second International Symposium on Operating Systems*, October 1978.

[25] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[26] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proc. NSDI'04*, March 2004.

[27] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[28] M. Melkonian. Get by Without an RTOS. *Embedded Systems Programming*, 13(10), September 2000.

[29] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *Proc. IPSN'05*, Los Angeles, CA, USA, April 2005.

[30] J. Paisley and J. Sventek. Real-time detection of grid bulk transfer traffic. In *Proceedings of the 10th IEEE/IFIP Network Operations Management Symposium*, 2006.

[31] J. C. Reynolds. The discoveries of continuations. *Lisp Symbol. Comput.*, 6(3):233–247, 1993.

[32] Z. B. Simpson. State machines: Cooperative mini-kernels with yielding. In *Computer Game Developer's Conference*, Austin, TX, November 1999.

[33] S. Tatham. Coroutines in C. Web page, 2000. http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

[34] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 171–180, 2003.

[35] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[36] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proc. SOSP '03*, pages 268–281, 2003.

[37] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. USENIX/ACM NSDI'04*, San Francisco, CA,, March 2004.