Protothreads

Lightweight, Stackless Threads in C

SICS Technical Report T2005:05
ISSN 1100-3154
ISRN:SICS-T–2005/05-SE
Keywords:

Multi-threading, event-driven programming, embedded systems.

Adam Dunkels
Swedish Institute of Computer Science
adam@sics.se

Oliver Schmidt
oliver@jantzer-schmidt.de

March 2005

Contents

Abstract

Protothreads are a extremely lightweight, stackless threads designed for use in severely memory constrained systems such as embedded systems. Software for memory constrained embedded systems sometimes are based on an event-driven model rather than on multi-threading. While event-driven systems allow for reduced memory usage, they require programs to be developed as explicit state machines. Since implementing programs as explicit state machines is hard, developing, maintaining, and debugging programs for event-driven systems is difficult.

Protothreads simplify implementation of high-level functionality on top of event-driven systems, without significantly increasing the memory requirements. Protothreads can be implemented in in the C programming language using 10 lines of code and 2 bytes of RAM per protothread.

Chapter 1

Protothreads

1.1 Introduction

Event-driven programming is a common concurrency mechanism for memory constrained embedded systems. Unlike multithreaded systems, event-driven systems do not require per-thread stacks thereby reducing the memory requirements. For this reason, we based both the uIP TCP/IP stack [7, 8] and the Contiki OS [6, 9] on an event-driven concurrency mechanism. However, after a few years of experience with programming these systems, it has become apparent that event-driven code is difficult to write, understand, maintain, and debug. This experience is similar to what others report [13].

While the event-driven model and the threaded model can be shown to be equivalent [12], programs written in the two models typically display differing characteristics [1]. The advantages and disadvantages of the two models are a debated topic [14, 18].

This report introduces *protothreads*, a novel concurrency mechanism intended to remedy some of the problems with event-driven programming, without the memory overhead of full multi-threading. We argue that protothreads can reduce the number of explicit state machines required to a type of programs that are typical for many embedded systems. We believe this reduction leads to programs that are easier to develop, debug, and maintain.

We demonstrate that protothreads can be implemented in the C programming language, using only standard C language constructs and without any architecture-specific machine code.

The rest of this report is structured as follows. Section 1.2 presents a motivating example and Section 1.3 introduces the notion of protothreads. Section 1.4 discusses related work, and the report is concluded in Section 1.5. Appendix A contains the full source code of the protothreads library, and Appendix B contains a condensed version of the code. Appendix C shows how a software driver for a particular radio interface chip was simplified using protothreads, and Appendix D includes the copyright license for all the source code in this report.

1.2 Motivation

To illustrate how high-level functionality is implemented using state machines, we consider a hypothetical energy-conservation mechanism for the nodes used in wireless sensor networks [2]. In such networks, it is very important that the nodes conserve their energy. Since the wireless communication device typically is the most energy consuming device, it is imperative that the communication device is turned off as often as possible to conserve the energy of the device.

Like most energy conservation mechanisms for wireless sensor networks, this hypothetical mechanism in this report switches the radio on and off at regular intervals. The mechanism works as follows:

1. Turn radio on.
2. Wait for t_{awake} milliseconds.
3. Turn radio off, but only if all communication has completed.
4. If communication has not completed, wait until it has completed. Then turn off the radio.
5. Wait for t_{sleep} milliseconds. If the radio could not be turned off before t_{sleep} milliseconds because of remaining communication, do not turn the radio off at all.
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 see three states: *on* – the radio is turned on, *waiting* – waiting for any remaining communication to complete, and *off* – the radio is off. Figure 1.1 shows the resulting state machine, including the state transitions.

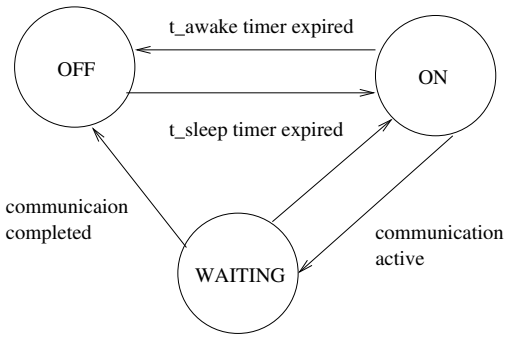

Figure 1.1: State machine realization of the radio sleep cycle protocol.

To implement this state machine in C, we use an explicit state variable, `state`, that can take on the values OFF, ON, and WAITING. We use a C switch statement to perform different actions depending on the `state` variable. The code is placed in an event handler function that is called whenever an event occurs. Possible events in this case are that a timer expires and that communication completes. The resulting C code is shown in Figure 1.2.

```
enum {
  ON,
  WAITING,
  OFF
} state;

void radio_wake_eventhandler() {
  switch(state) {

  case OFF:
    if(timer_expired(&timer)) {
      radio_on();
      state = ON;
      timer_set(&timer, T_AWAKE);
    }
    break;

  case ON:
    if(timer_expired(&timer)) {
      timer_set(&timer, T_SLEEP);
      if(!communication_complete()) {
        state = WAITING;
      } else {
        radio_off();
        state = OFF;
      }
    }
    break;

  case WAITING:
    if(communication_complete()
       || timer_expired(&timer)) {
      state = ON;
      timer_set(&timer, T_AWAKE);
    } else {
      radio_off();
      state = OFF;
    }
    break;
  }
}
```

Figure 1.2: The radio sleep cycle implemented with events.

We note that this simple mechanism results in a fairly large amount of C code. The structure of the mechanism, as it is described by the six steps above, is not immediately evident from the C code.

1.3 Protothreads

Protothreads [5] are an extremely lightweight stackless type of threads, designed for severely memory constrained systems. Protothreads provide *conditional blocking* on top of an event-driven system, without the overhead of per-thread stacks.

We developed protothreads in order to deal with the complexity of explicit state machines in the event-driven uIP TCP/IP stack [8]. For uIP, we were able to substan-

```
PT_THREAD(radio_wake_thread(struct pt *pt)) {
  PT_BEGIN(pt);

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);
    PT_WAIT_UNTIL(pt, timer_expired(&timer));

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
      PT_WAIT_UNTIL(pt, communication_complete()
                        || timer_expired(&timer));
    }

    if(!timer_expired(&timer)) {
      radio_off();
      PT_WAIT_UNTIL(pt, timer_expired(&timer));
    }
  }

  PT_END(pt);
}
```

Figure 1.3: The radio sleep cycle implemented with protothreads.

tially reduce the number of state machines and explicit states used in the implementations of a number of application level communication protocols. For example, the uIP FTP client could be simplified by completely removing the explicit state machine, and thereby reducing the number of explicit states from 20 to one.

1.3.1 Protothreads versus events

Programs written for an event-driven model typically have to be implemented as explicit state machines. In contrast, with protothreads programs can be written in a sequential fashion without having to design explicit state machines. To illustrate this, we return to the radio sleep cycle example from the previous section.

Figure 1.3 shows how the radio sleep cycle mechanism is implemented with protothreads. Comparing Figure 1.3 and Figure 1.2, we see that the protothreads-based implementation not only is shorter, but also more closely follows the specification of the radio sleep mechanism. Due to the linear code flow of this implementation, the overall logic of the sleep cycle mechanism is visible in the C code. Also, in the protothreads-based implementation we are able to make use of regular C control flow mechanisms such as while loops and if statements.

1.3.2 Protothreads versus threads

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 stack for a thread might use a large part of the available memory. In comparison, the memory requirements of a protothread that of an unsigned integer. No additional stack is needed for the protothread.

Unlike a thread, a protothread runs only within a single C function and cannot span over other functions. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls is instead implemented by spawning a separate protothread for each potentially blocking function. Unlike threads, protothreads makes blocking explicit: the programmer knows exactly which functions that potentially may yield.

1.3.3 Comparison

Table 1.1 summarizes the features of protothreads and compares them with the features of events and threads.

Feature	Events	Threads	**Proto-threads**
Control structures	No	Yes	**Yes**
Debug stack retained	No	Yes	**Yes**
Implicit locking	Yes	No	**Yes**
Preemption	No	Yes	**No**
Automatic variables	No	Yes	**No**

Table 1.1: Qualitative comparison between events, threads and protothreads

Control structures. One of the advantages of threads over events is that threads allow programs to make full use of the control structures (e.g., *if* conditionals and *while* loops) provided by the programming language. In the event-driven model, control structures must be broken down into two or more pieces in order to implement continuations [1]. In contrast, both threads and protothreads allow blocking statements to be used together with control structures.

Debug stack retained. Because the manual stack management and the free flow of control in the event-driven model, debugging is difficult as the sequence of calls is not saved on the stack [1]. With both threads and protothreads, the full call stack is available for debugging.

Implicit locking. With manual stack management, as in the event-driven model, all yield points are immediately visible in the code. This makes it evident to the programmer whether or not a structure needs to be locked. In the threaded model, it is not as evident that a particular function call yields. Using protothreads, however, potentially blocking statements are explicitly implemented with a `PT_WAIT` statement. Program code between such statements never yields.

Preemption. The semantics of the threaded model allows for preemption of a running thread: the thread's stack is saved, and execution of another thread can be continued. Because both the event-driven model and protothreads use a single stack, preemption is not possible within either of these models.

Automatic variables. Since the threaded model allocates a stack for each thread, automatic variables—variables with function local scope automatically allocated on the stack—are retained even when the thread blocks. Both the event-driven model and protothreads use a single shared stack for all active programs, and

rewind the stack every time a program blocks. Therefore, with protothreads, automatic variables are not saved across a blocking wait. This is discussed in more detail below.

1.3.4 Limitations

While protothreads allow programs to take advantage of a number of benefits of the threaded programming model, protothreads also impose some of the limitations from the event-driven model. The most evident limitation from the event-driven model is that automatic variables—variables with function-local scope that are automatically allocated on the stack—are not saved 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. The reason for this is that protothreads rewind the stack at every blocking statement, and therefore potentially destroy the contents of variables on the stack.

Many optimizing C compilers, including gcc, are able to detect if an automatic variable is unsafely used after a blocking statement. Typically a warning is produced, stating that the variable in question "might be used uninitialized in this function". While it may not be immediately apparent for the programmer that this warning is related to the use of automatic variables across a blocking protothreads statement, it does provide an indication that there is a problem with the program. Also, the warning indicates the line number of the problem which assists the programmer in identifying the problem.

The limitation on the use of automatic variables can be handled by using an explicit *state object*, much in the same way as is done in the event-driven model. The state object is a chunk of memory that holds the contents of all automatic variables that need to be saved across a blocking statement. It is, however, the responsibility of the programmer to allocate and maintain such a state object.

It should also be noted that protothreads do not limit the use of *static local* variables. Static local variables are variables that are local in scope but allocated in the data section. Since these 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 re-entrant, using static local variables instead of automatic variables can be an acceptable solution to the problem.

1.3.5 Implementation

Protothreads are based on a low-level mechanism that we call *local continuations* [10]. A local continuation is similar to ordinary continuations [15], but does not capture the program stack. Local continuations can be implemented in a variety of ways, including using architecture specific machine code, C-compiler extensions, and a non-obvious use of the C *switch* statement. In this paper, we concentrate on the method based on the C switch statement.

A local continuation supports two operations; it can be either *set* or *resumed*. When a local continuation is set, the state of the function—all CPU registers including the program counter but excluding the stack—is captured. When the same local continuation is resumed, the state of the function is reset to what it was when the local continuation was set.

A protothread consists of a C function and a single local continuation. The protothread's local continuation is *set* before each conditional blocking wait. If the condi-

tion is true and the wait is to be performed, the protothread executes an explicit return statement, thus returning to the caller. The next time the protothread is invoked, the protothread *resumes* the local continuation that was previously set. This will effectively cause the program to jump to the conditional blocking wait statement. The condition is re-evaluated and, once the condition is false, the protothread continues to execute the function.

```
#define RESUME(lc) switch(lc) { case 0:
```

```
#define SET(lc)    lc = __LINE__; case __LINE__:
```

Figure 1.4: The local continuation *resume* and *set* operations implemented using the C switch statement.

Local continuations can be implemented using standard C language constructs and a non-obvious use of the C switch statement. With this technique, the local continuation is represented by an unsigned integer. The resume operation is implemented as an open switch statement, and the set operation is implemented as an assignment of the local continuation and a case statement, as shown in Figure 1.4. Each set operation sets the local continuation to a value that is unique within each function, and the resume operation's switch statement jumps to the corresponding case statement. The case 0: statement in the implementation of the resume operation ensures that the resume statement does nothing if is the local continuation is zero.

Figure 1.5 shows the example radio sleep cycle mechanism from Section 1.2 with the protothreads statements expanded using the C switch implementation of local continuations. We see how each `PT_WAIT_UNTIL` statement has been replaced with a case statement, and how the `PT_BEGIN` statement has been replaced with a switch statement. Finally, the `PT_END` statement has been replaced with a single right curly bracket, which closes the switch block that was opened by the `PT_BEGIN` statement. We also note the similarity between Figure 1.5 and the event-based implementation in Figure 1.2. While the resulting C code is very similar in the two cases, the process of arriving at the code is different. With the event-driven model, the programmer must explicitly design and implement a state machine. With protothreads, the state machine is automatically generated.

The non-obviousness of the C switch implementation of local continuations is that the technique appears to cause problems when a conditional blocking statement is used inside a nested C control statement. For example, the case 13: statement in Figure 1.5 appears inside an if block, while the corresponding switch statement is located at a higher block. However, this is a valid use of the C switch statement: case statements may be located anywhere inside a switch block. They do not need to be in the same level of nesting, but can be located anywhere, even inside nested if or for blocks. This use of the switch statement is likely to first have been publicly described by Duff [4]. The same technique has later been used by Tatham to implement coroutines in C [16].

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 is 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 imple-

```
void radio_wake_thread(struct pt *pt) {
  switch(pt->lc) {
  case 0:

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);

    pt->lc = 8;
  case 8:
    if(!timer_expired(&timer)) {
      return;
    }

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {

      pt->lc = 13;
  case 13:
      if(!(communication_complete() ||
           timer_expired(&timer))) {
        return;
      }

    }

    if(!timer_expired(&timer)) {
      radio_off();

      pt->lc = 18;
  case 18:
      if(!timer_expired(&timer)) {
        return;
      }

    }
  }
  }
}
```

Figure 1.5: C switch statement expansion of the protothreads code in Figure 1.3

mentation of protothreads and switch statements. We have not yet found a suitable solution for this problem.

1.4 Related Work

Kasten and Römer [11] have also identified the need for new abstractions for managing the complexity of event-triggered programming. They introduce OSM, a state machine programming model based on Harel's StateCharts. The model reduces both the complexity of the implementations and the memory usage. Their work is different from protothreads in that OSM requires support from an external OSM compiler to produce the resulting C code, whereas protothreads only make use of the regular C preprocessor.

Dabek et. al [3] presents *libasynch*, a C++ library that assist 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 flow control statements across blocking calls.

Adya et. al [1] discusses the respective merits of event-driven and threaded programming and presents 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 of which approach the callee uses.

1.5 Conclusions

Event-driven programming is a common concurrency model for memory constrained embedded systems. In order to implement high-level operations under this model, programs have to be written as explicit state machines. Software implemented using explicit state machines is often hard to understand, debug, and maintain.

We have presented *protothreads* as a programming abstraction that reduces the complexity of implementations of high-level functionality for event-triggered systems. With protothreads, programs can perform *conditional blocking* on top of event-triggered systems with run-to-completion semantics, without the overhead of full multi-threading.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European Commission under contract IST-004536-RUNES.

Bibliography

[1] 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*, pages 289–302, 2002.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Commun. Mag.*, 40(8):102–114, 2002.

[3] 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.

[4] T. Duff. Re: Explanation please! Usenet news article, Message-ID: <8144@alice.UUCP>, August 1988.

[5] A. Dunkels. Protothreads web site. Web page. Visited 2005-03-18.
URL: http://www.sics.se/~adam/pt/

[6] A. Dunkels. The Contiki Operating System. Web page. Visited 2005-03-18.
URL: http://www.sics.se/~adam/contiki/

[7] A. Dunkels. uIP - a TCP/IP stack for 8- and 16-bit microcontrollers. Web page. Visited 2005-03-18.
URL: http://www.sics.se/~adam/uip/

[8] 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.

[9] 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.

[10] A. Dunkels and O. Schmidt. Protothreads – Lightweight Stackless Threads in C. Technical Report T2005:05, Swedish Institute of Computer Science.

[11] 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.

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

[13] 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.

[14] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference, 1996.

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

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

[17] The GCC Team. The GNU compiler collection. Web page. 2002-10-14.
URL: http://gcc.gnu.org/

[18] 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.

Appendix A

The Protothreads Library Source Code

This appendix presents the full implementation of the protothreads library, as well as two implementations of local continuations.

A.1 Protothreads

```
struct pt {
  lc_t lc;
};

#define PT_THREAD_WAITING 0
#define PT_THREAD_EXITED  1

/**
 * Declaration of a protothread.
 *
 * This macro is used to declare a protothread. All protothreads must
 * be declared with this macro.
 *
 */
#define PT_THREAD(name_args) char name_args

/**
 * Initialize a protothread.
 *
 * Initializes a protothread. Initialization must be done prior to
 * starting to execute the protothread.
 *
 */
#define PT_INIT(pt)  LC_INIT((pt)->lc)

/**
 * Declare the start of a protothread inside the C function
 * implementing the protothread.
 *
 * This macro is used to declare the starting point of a
 * protothread. It should be placed at the start of the function in
 * which the protothread runs. All C statements above the PT_BEGIN()
 * invokation will be executed each time the protothread is scheduled.
 *
```

```
 */
#define PT_BEGIN(pt) LC_RESUME((pt)->lc)
/*\
  do { \
    if((pt)->lc != LC_NULL) { \
      LC_RESUME((pt)->lc); \
    }  \
    } while(0)*/

/**
 * Block and wait until condition is true.
 *
 * This macro blocks the protothread until the specified condition is
 * true.
 *
 */
#define PT_WAIT_UNTIL(pt, condition)         \
  do { \
    LC_SET((pt)->lc); \
    if(!(condition)) { \
      return PT_THREAD_WAITING; \
    } \
  } while(0)

/**
 * Block and wait while condition is true.
 *
 * This function blocks and waits while condition is true. See
 * PT_WAIT_UNTIL().
 *
 */
#define PT_WAIT_WHILE(pt, cond) PT_WAIT_UNTIL((pt), !(cond))

/**
 * Block and wait until a child protothread completes.
 *
 * This macro schedules a child protothread. The current protothread
 * will block until the child protothread completes.
 *
 * \note The child protothread must be manually initialized with the
 * PT_INIT() function before this function is used.
 *
 */
#define PT_WAIT_THREAD(pt, thread) PT_WAIT_WHILE((pt), PT_SCHEDULE(thread))

/**
 * Spawn a child protothread and wait until it exits.
 *
 * This macro spawns a child protothread and waits until it exits. The
 * macro can only be used within a protothread.
 *
 */
#define PT_SPAWN(pt, pt2, thread) \
  do { \
    PT_INIT((pt2)); \
    PT_WAIT_THREAD((pt), (thread)); \
  } while(0)

/**
 * Restart the protothread.
 *
 * This macro will block and cause the running protothread to restart
```

```
 * its execution at the place of the PT_BEGIN() call.
 *
 */
#define PT_RESTART(pt) \
  do { \
    PT_INIT(pt); \
    return PT_THREAD_WAITING; \
  } while(0)

/**
 * Exit the protothread.
 *
 * This macro causes the protothread to exit. If the protothread was
 * spawned by another protothread, the parent protothread will become
 * unblocked and can continue to run.
 *
 */
#define PT_EXIT(pt) \
  do { \
    PT_INIT(pt); \
    return PT_THREAD_EXITED; \
  } while(0)

/**
 * Declare the end of a protothread.
 *
 * This macro is used for declaring that a protothread ends. It should
 * always be used together with a matching PT_BEGIN() macro.
 *
 */
#define PT_END(pt) LC_END((pt)->lc); PT_EXIT(pt)


/**
 * Schedule a protothread.
 *
 * This function shedules a protothread. The return value of the
 * function is non-zero if the protothread is running or zero if the
 * protothread has exited.
 *
 */
#define PT_SCHEDULE(f) (f == PT_THREAD_WAITING)
```

A.2 Local continuations based on the C switch() statement

This implementation of local continuations uses the C switch() statement to resume execution of a function somewhere inside the function's body. The implementation is based on the fact that switch() statements are able to jump directly into the bodies of control structures such as if() or while() statements.

This implementation is heavily inspired by Simon Tatham's coroutines implementation in C [16].

```
typedef unsigned short lc_t;

#define LC_INIT(s) s = 0;

#define LC_RESUME(s) switch(s) { case 0:

#define LC_SET(s) case __LINE__: s = __LINE__

#define LC_END(s) }
```

A.3 Local continuations based on address labels

This implementation of local continuations is based on a special feature of the GCC C compiler called "labels as values". This feature allows assigning pointers with the address of the code corresponding to a particular C label. The GCC documentation [17] comments the "labels as value" function as follows:

> The analogous feature in Fortran is called an assigned goto, but that name seems inappropriate in C, where one can do more than simply store label addresses in label variables.

This implementation constructs a label for each `LC_SET()` statement and stores the address of the label in the `lc_t` variable. The label is constructed by concatenating the line number of the `LC_SET()` statement with the string `LC_LABEL`. Due to specifics of the C preprocessor, the concatenation must be done using two separate macros, `LC_CONCAT1()` and `LC_CONCAT2()`.

```
typedef void * lc_t;

#define LC_INIT(s) s = NULL

#define LC_CONCAT2(s1, s2) s1##s2
#define LC_CONCAT(s1, s2) LC_CONCAT2(s1, s2)

#define LC_RESUME(s) \
  do { \
    if(s != NULL) { \
      goto *s; \
    } \
  } while(0)

#define LC_SET(s) \
  do { \
    LC_CONCAT(LC_LABEL, __LINE__):            \
    (s) = &&LC_CONCAT(LC_LABEL, __LINE__); \
  } while(0)

#define LC_END(s)
```

Appendix B

The Condensed Source Code

This appendix presents an implementation of protothreads where the protothreads functions have been collapsed with the local continuations implementation based on the C switch() trick described in Section 1.3.5. This implementation consists of only seven lines of fully portable C code.

```
struct pt { unsigned short lc; };
#define PT_INIT(pt)           (pt)->lc = 0
#define PT_SCHEDULE(f)        (f)
#define PT_THREAD(name_args) char name_args
#define PT_BEGIN(pt)          switch((pt)->lc) { case 0:
#define PT_WAIT_UNTIL(pt, c) (pt)->lc = __LINE__; case __LINE__: \
                              if(!(c)) return 1
#define PT_EXIT(pt)           (pt)->lc = 0; return 0
#define PT_END(pt)            } (pt)->lc = 0; return 0
```

Appendix C

Example: TR1001 Input Driver

This appendix contains the implementation of an interrupt handler of a device driver for a TR1001 radio module. The driver receives incoming data in bytes and constructs a frame that is covered by a CRC checksum. The driver is implemented both with protothreads and with an explicit state machine. The state machine has 11 states and is implemented using the C switch() statement.

The flow of control in the state machine-based implementation is quite hard to follow from inspection of the code, whereas the flow of control is evident in the protothreads based implementation.

C.1 Protothreads-based implementation

```
PT_THREAD(tr1001_rxhandler(unsigned char incoming_byte))
{
  PT_YIELDING();
  static unsigned char rxtmp, tmppos;

  PT_BEGIN(&rxhandler_pt);

  while(1) {

    /* Wait until we receive the first syncronization byte. */
    PT_WAIT_UNTIL(&rxhandler_pt, incoming_byte == SYNCH1);

    tr1001_rxstate = RXSTATE_RECEVING;

    /* Read all incoming syncronization bytes. */
    PT_WAIT_WHILE(&rxhandler_pt, incoming_byte == SYNCH1);

    /* We should receive the second synch byte by now, otherwise we'll
       restart the protothread. */
    if(incoming_byte != SYNCH2) {
      PT_RESTART(&rxhandler_pt);
    }

    /* Reset the CRC. */
    rxcrc = 0xffff;

    /* Read packet header. */
    for(tmppos = 0; tmppos < TR1001_HDRLEN; ++tmppos) {
```

```
  /* Wait for the first byte of the packet to arrive. */
  PT_YIELD(&rxhandler_pt);

  /* If the incoming byte isn't a valid Manchester encoded byte,
     we start again from the beinning. */
  if(!me_valid(incoming_byte)) {
    PT_RESTART(&rxhandler_pt);
  }

  rxtmp = me_decode8(incoming_byte);

  /* Wait for the next byte to arrive. */
  PT_YIELD(&rxhandler_pt);

  if(!me_valid(incoming_byte)) {
    PT_RESTART(&rxhandler_pt);
  }

  /* Put together the two bytes into a single Manchester decoded
     byte. */

  tr1001_rxbuf[tmppos] = (rxtmp << 4) | me_decode8(incoming_byte);

  /* Calculate the CRC. */
  rxcrc = crc16_add(tr1001_rxbuf[tmppos], rxcrc);


}

/* Since we've got the header, we can grab the length from it. */
tr1001_rxlen = ((((struct tr1001_hdr *)tr1001_rxbuf)->len[0] << 8) +
                ((struct tr1001_hdr *)tr1001_rxbuf)->len[1]);


/* If the length is longer than we can handle, we'll start from
   the beginning. */
if(tmppos + tr1001_rxlen > sizeof(tr1001_rxbuf)) {
  PT_RESTART(&rxhandler_pt);
}

/* Read packet data. */
for(tmppos = 6; tmppos < tr1001_rxlen + TR1001_HDRLEN; ++tmppos) {

  PT_YIELD(&rxhandler_pt);

  if(!me_valid(incoming_byte)) {
    PT_RESTART(&rxhandler_pt);
  }

  rxtmp = me_decode8(incoming_byte);

  PT_YIELD(&rxhandler_pt);

  if(!me_valid(incoming_byte)) {
    PT_RESTART(&rxhandler_pt);
  }

  tr1001_rxbuf[tmppos] = (rxtmp << 4) | me_decode8(incoming_byte);
  rxcrc = crc16_add(tr1001_rxbuf[tmppos], rxcrc);

}
```

```
    /* Read the frame CRC. */
    for(tmppos = 0; tmppos < 4; ++tmppos) {

      PT_YIELD(&rxhandler_pt);

      if(!me_valid(incoming_byte)) {
        PT_RESTART(&rxhandler_pt);
      }

      rxcrctmp = (rxcrctmp << 4) | me_decode8(incoming_byte);
    }

    if(rxcrctmp == rxcrc) {
      /* A full packet has been received and the CRC checks out. We'll
         request the driver to take care of the incoming data. */

      tr1001_drv_request_poll();

      /* We'll set the receive state flag to signal that a full frame
         is present in the buffer, and we'll wait until the buffer has
         been taken care of. */
      tr1001_rxstate = RXSTATE_FULL;
      PT_WAIT_UNTIL(&rxhandler_pt, tr1001_rxstate != RXSTATE_FULL);
    }
  }
  PT_END(&rxhandler_pt);
}
```

C.2 State machine-based implementation

```
/* No bytes read, waiting for synch byte. */
#define RXSTATE_READY   0
/* Second start byte read, waiting for header. */
#define RXSTATE_START   1
/* Reading packet header, first Manchester encoded byte. */
#define RXSTATE_HEADER1 2
/* Reading packet header, second Manchester encoded byte. */
#define RXSTATE_HEADER2 3
/* Reading packet data, first Manchester encoded byte. */
#define RXSTATE_DATA1   4
/* Reading packet data, second Manchester encoded byte. */
#define RXSTATE_DATA2   5
/* Receiving CRC16 */
#define RXSTATE_CRC1    6
#define RXSTATE_CRC2    7
#define RXSTATE_CRC3    8
#define RXSTATE_CRC4    9
/* A full packet has been received. */
#define RXSTATE_FULL    10


void
tr1001_rxhandler(unsigned char c)
{
  switch(tr1001_rxstate) {
  case RXSTATE_READY:
    if(c == SYNCH1) {
      tr1001_rxstate = RXSTATE_START;
      rxpos = 0;
    }
    break;
```

```
case RXSTATE_START:
  if(c == SYNCH1) {
    tr1001_rxstate = RXSTATE_START;
  } else if(c == SYNCH2) {
    tr1001_rxstate = RXSTATE_HEADER1;
    rxcrc = 0xffff;
  } else {
    tr1001_rxstate = RXSTATE_READY;
  }
  break;
case RXSTATE_HEADER1:
  if(me_valid(c)) {
    tr1001_rxbuf[rxpos] = me_decode8(c);
    tr1001_rxstate = RXSTATE_HEADER2;
  } else {
    tr1001_rxstate = RXSTATE_ERROR;
  }
  break;
case RXSTATE_HEADER2:
  if(me_valid(c)) {
    tr1001_rxbuf[rxpos] = (tr1001_rxbuf[rxpos] << 4) |
      me_decode8(c);
    rxcrc = crc16_add(tr1001_rxbuf[rxpos], rxcrc);

    ++rxpos;
    if(rxpos >= TR1001_HDRLEN) {
      tr1001_rxlen = ((((struct tr1001_hdr *)tr1001_rxbuf)->len[0] << 8) +
                ((struct tr1001_hdr *)tr1001_rxbuf)->len[1]);
      if(rxpos + tr1001_rxlen == sizeof(tr1001_rxbuf)) {
        tr1001_rxstate = RXSTATE_DATA1;
      }
    } else {
      tr1001_rxstate = RXSTATE_HEADER1;
    }
  } else {
    tr1001_rxstate = RXSTATE_READY;
  }
  break;
case RXSTATE_DATA1:
  if(me_valid(c)) {
    tr1001_rxbuf[rxpos] = me_decode8(c);
    tr1001_rxstate = RXSTATE_DATA2;
  } else {
    tr1001_rxstate = RXSTATE_READY;
  }
  break;
case RXSTATE_DATA2:
  if(me_valid(c)) {
    tr1001_rxbuf[rxpos] = (tr1001_rxbuf[rxpos] << 4) |
      me_decode8(c);

    rxcrc = crc16_add(tr1001_rxbuf[rxpos], rxcrc);

    ++rxpos;
    if(rxpos == tr1001_rxlen + TR1001_HDRLEN) {
      tr1001_rxstate = RXSTATE_CRC1;
    } else if(rxpos > sizeof(tr1001_rxbuf)) {
      tr1001_rxstate = RXSTATE_READY;
    } else {
      tr1001_rxstate = RXSTATE_DATA1;
    }
  } else {
```

```
      tr1001_rxstate = RXSTATE_READY;
    }
    break;
  case RXSTATE_CRC1:
    if(me_valid(c)) {
      rxcrctmp = me_decode8(c);
      tr1001_rxstate = RXSTATE_CRC2;
    } else {
      tr1001_rxstate = RXSTATE_READY;
    }
    break;
  case RXSTATE_CRC2:
    if(me_valid(c)) {
      rxcrctmp = (rxcrctmp << 4) | me_decode8(c);
      tr1001_rxstate = RXSTATE_CRC3;
    } else {
      tr1001_rxstate = RXSTATE_READY;
    }
    break;
  case RXSTATE_CRC3:
    if(me_valid(c)) {
      rxcrctmp = (rxcrctmp << 4) | me_decode8(c);
      tr1001_rxstate = RXSTATE_CRC4;
    } else {
      tr1001_rxstate = RXSTATE_READY;
    }
    break;
  case RXSTATE_CRC4:
    if(me_valid(c)) {
      rxcrctmp = (rxcrctmp << 4) | me_decode8(c);
      if(rxcrctmp == rxcrc) {
        tr1001_rxstate = RXSTATE_FULL;
        tr1001_drv_request_poll();
      } else {
        tr1001_rxstate = RXSTATE_READY;
      }
    } else {
      tr1001_rxstate = RXSTATE_READY;
    }
    break;
  case RXSTATE_FULL:
    /* Just drop the incoming byte. */
    break;
  default:
    /* Just drop the incoming byte. */
    tr1001_rxstate = RXSTATE_READY;
    break;
  }
}
```

Appendix D

Copyright License

This source code presented in this report is protected by copyright. The source code is, however, available under the open source copyright license below.

Copyright (c) 2004-2005, Swedish Institute of Computer Science. All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the Institute nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Author: Adam Dunkels <adam@sics.se>