Use of Forth to Enable Distributed Processing on Wireless Sensor Networks

S. Gaglio, G. Lo Re, G. Martorella, D. Peri

In Proceedings of the 31th EuroForth Conference (EuroForth2015)

Use of Forth to Enable Distributed Processing on Wireless Sensor Networks

Salvatore Gaglio
salvatore.gaglio@unipa.it

Giuseppe Lo Re
giuseppe.lore@unipa.it

Gloria Martorella
gloria.martorella@unipa.it

Daniele Peri
daniele.peri@unipa.it

Abstract

Wireless Sensor Networks (WSNs) are composed of tiny sensor nodes able to monitor environmental conditions. Existing applications for WSNs usually adopt a centralized approach that exploit sensor nodes just for sensing, while data processing takes place on more powerful base stations. This can be considered a consequence of the common WSN programming practice that proves too rigid to support development based on distributed processing. In fact, local processing of complex data, such as symbolic information and rules, is an under explored aspect. The adoption of high level interpreters above general purpose operating systems is often unpractical since it implies the saturation of the available resources. In this paper, we detail the implementation of an alternative Forth-based approach that implements a minimal but extensible operating system featuring common WSN functionalities as well as advanced skills such as symbolic distributed processing. We show the definition of words and syntactic constructs that enable collaborative processing on WSNs and ease the development of complex applications even on resource constrained WSN nodes. To this purpose, our approach is based on an abstract mechanism enabling nodes to exchange directly Forth code. Cooperative behaviors, introducing dynamic computation into the network, are thus easily implemented, as we show in a few applicative examples. Moreover, using the same mechanism, remote nodes can be effortlessly reprogrammed even after their deployment. Finally, we show how our approach proves to be feasible and advantageous through a comparison, in terms of memory usage, with relevant interpreter-based software platforms for WSNs.

1 Introduction

Wireless Sensor Networks (WSNs) are composed of tiny wirelessly interconnected sensor nodes that are equipped with a microcontroller, a radio interface subsystem, some sensor devices and an autonomous power supply, usually consisting in batteries [1]. Generally, such devices are characterized by quite constrained resources in terms of energy, communication and processing capabilities.

WSNs represent a very active research area as several applications have been proposed in literature in several contexts such as biomedical, healthcare, military, industrial and environmental fields [2].

The development of high level applications is typically supported by general purpose operating systems for WSNs such as Contiki and TinyOS [3], which primarily focus on reducing power consumptions while optimizing resource usage [4].

Mainstream programming practices involve the cross-compilation of specialized code with the thin layer operating system of choice, and the subsequent code uploading to the on-board ROM memory. Any modifications in the source code lead to retrace the same steps afresh.

Such practice strongly limits the development of more advanced applications than the static acquisition and transmission of sensory data that is then to be processed by a base station [5].

Sophisticated applications, such as those concerning Ambient Intelligence (AmI) scenarios, could instead be developed if the nodes were able to process cooperatively more complex data –e.g. symbolic data and rules– than the numerical values in rigid representations resulting from sensing. Such applications may in fact implement intelligent, autonomic, and self-organizing behaviors by distributed processing of symbolic and qualitative description of the observed phenomena. However, due to memory constraints of the available development methodologies, such kind of applications are too complex to be implemented on WSNs without recurring to centralized or Cloud-based infrastructures [6].

In order to give the network some adaptivity to changes of the environment as well as of the application goals after the nodes have been deployed, alternative WSN application development tools are thus strongly required [7].

To overcome the inflexibility of conventional programming methodologies, several interpreters targeting resource constrained Wireless Sensor Network (WSN) nodes have been presented in literature [8], [9], [6], [10]. Their primary goal is to support the application development as well as the retasking of already deployed nodes. However, node reprogramming affects just the application code, while the hardware-abstraction layer modifications require to upload the whole binary image or to replace just the modules to be updated [11].

In general, high-level language interpreters are designed as applications running atop the chosen general purpose operating system. Unfortunately, such a strategy dramatically increases the processing load on the on-board microcontrollers, and detaches the application from the hardware. Moreover, this solution often leads to high memory occupation that leaves insufficient memory resources to develop not trivial applications [8].

The choice of Forth in WSN AmI applications, which are characterized by realtime and resource constraints, seems thus quite natural and desirable [12]. Moreover, the interactive nature of Forth makes it easy to face the challenges of AmI development with experimental programming.

In this paper, we detail our experimentation on the use of Forth on WSN nodes as an operating system and development tool. We describe our ongoing implementation of a Forth-based software platform that provides nodes with basic WSN capabilities such as networking, sensing, and actuation, accessible through expressive words, and easily extensible to support complex functionalities.

Distributed processing is one of the key goals of our platform that we addressed with a simple abstract mechanism based on the transmission of Forth code among nodes, even already deployed ones. In the next sections, we show how we have been able to implement this abstraction in a few dozen words, with a remarkably low resource usage with respect to other available interpreters.

The remainder of the paper is organized as follows. Section 2 details the wordset we have implemented to use Forth as an operating system for WSNs. In Section 3, the primitives supporting distributed processing are presented. Section 4 describes some working applications running on WSN nodes in order to demonstrate the feasibility of our approach and finally Section 5 reports our conclusions.

2 Forth as an Operating System for WSNs

Forth naturally provides an interactive environment with most of the functionalities of an operating system for common computers. In the case of WSN nodes the OS responsibilities include the management of networking as well as all the various on-board and optional sensors and devices.

Most WSN nodes –referred to as *motes* in the specific literature– are based on MCU with Harvard architecture with separate memories for data and programs. Several interfaces, e.g. digital I/O, analog inputs, I2C, SPI and UARTs enable the connection with external modules, such as the radio subsystem, sensing boards, and so forth. For instance, the IrisMote platform that we used as a testbed, which is one of the most adopted, especially for research purposes [13], is equipped with an IEEE 802.15.4 compliant radio transceiver, 128 KB of Flash memory, 8 KB of static RAM and a 4 KB EEPROM, and can be expanded with sensing and prototyping boards.

At the beginning of our experimentation, we sorted out all the available Forth environments targeting the AVR microcontroller used in the IrisMote platform. We chose AmForth [14], a simple indirect threaded code interpreter, as it proved mature enough to be used as a development tool, and as it also provided a usable interactive shell through a serial terminal. However, AmForth did not include natively the support for any WSN platform. This required us to patch AmForth for the IrisMote to include specific configuration settings, such as those concerning ports, clock generators, on-board radio registers, timers and so on.

In our efforts to build an operating system for WSNs we defined an essential collection of definitions for the basic functionalities needed by WSN applications, such as networking and sensing.

Not all the definitions are strictly related to the hardware. Instead, we defined some more generic and platform-independent words that are not tied to specific hardware implementation and can thus easily work on different platforms. As an example, hardware independent words are those used to create valid data frames according to the 802.15.4 standard. In our implementation, transmission and reception of valid 802.15.4 frames is based on two buffers:

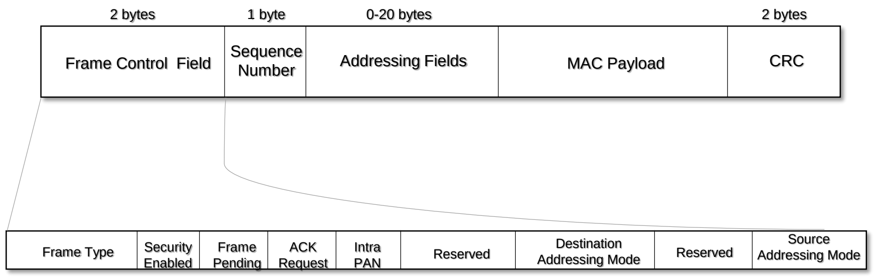

Figure 1: Format of a valid MAC layer frame according to the 802.15.4 standard. The frame control field is detailed in the bottom of the figure.

- outbound: a memory area where the outgoing frame is stored before downloading it to the radio frame buffer for the transmission;
- inbound: a memory area where the received frame is stored after it is uploaded from the radio frame buffer.

The buffers are 128 bytes long. According to the 802.15.4-2003 standard (see Figure 1), we defined the words to create valid data frames and to set the frame fields appropriately, e.g. short/long destination addressing mode field, frame type, frame length, and so on. In particular, Listing 1 shows the word definition to create a default frame with the following settings:

- short addressing mode (source and destination);
- intra-pan bit set to 1;
- 0xabcd pan address;

Listing 1: Forth word to create a valid 802.15.4 data frame with fixed settings

```
: default-pkt ( -- packet )
outbound dup erase
data frame_type
pan_compr
dest short mode! src short mode!
dest pan $abcd s_addr! src addr id @ s_addr! ;
```

2.1 Forth Words for Input Redirection to the Radio Module

Our Forth-based implementation supports interactive development on already deployed devices. This feature permits adding new words on remote nodes even if they are not physically connected to a serial terminal. Interactivity, symbolic processing and executable code exchange are the pivotal characteristics of our system.

The code exchanged among nodes and received from the radio channel is interpreted by the system, provided that the default input –the USART, at boot– has been redirected to the radio transceiver. Each incoming frame triggers the interrupt invoking the text interpreter on the frame payload.

Listing 2 shows the word definitions to enable the interpretation of incoming frames from the radio channel, by switching the input from the USART to the radio.

Listing 2: Forth words to redirect the standard input device to the radio

```
variable old-key
variable old-key?
variable payld-addr
variable payld-size
variable payld-in
$200 constant usart_rx_in
$201 constant usart_rx_out
$202 constant usart_rx_data

: payld-reset
    0 payld-size !
    0 payld-in ! ;

: payld-set ( addr n -- )
    payld-size !
    payld-addr !
    0 payld-in ! ;

: radio-key?
    payld-size @ dup 0 > swap
    payld-in @ > and ;

: radio-keyin
    payld-in @ dup payld-addr @ +
    c@ swap 1 + payld-in ! ;

: radio-key
  begin pause radio-key? until radio-keyin ;


: +radio-input
  payld-reset
  [’] key defer@ old-key !
  [’] key? defer@ old-key? !
  [’] radio-key is key
  [’] radio-key? is key? ;

: -radio-input
  old-key @ is key
  old-key? @ is key? ;

: usart_inject
  usart_rx_in c@ usart_rx_data + !
  1 usart_rx_in +! ;
```

Essentially, the input redirection makes the deferred words `key?` and `key` point to `radio-key?` and `radio-key` respectively. The word `radio-key?` is used to assess if there are unread characters in the frame payload by checking either if the variable `payld-size` is greater than 0 and the current pointer to the payload `payld-in` is lower than `payld-size`. The word `radio-keyin` fetches the next character in the frame payload and advances the current payload pointer `payld-in`. Finally, the word `radio-key` executes `radio-key?` and `radio-keyin` until all the characters in the frame payload have been read. To redirect the input to the radio, the word `+radio-input` is typed in the node shell. The execution causes the AmForth shell to be lost, until a data frame containing the word `-radio-input` is received. This word restores the input to the USART.

Code processing takes place directly in the interpreter loop as the last character of each incoming frame is required to be a carriage return. Such an event triggers the interpretation of the payload. However, in real use, interacting with networked devices through a wired line is unpractical. In fact, to redirect the input to the radio system without any wired connection, we defined a special frame containing just the character $17, which is the ASCII code for the non-printable character ETB.

Once a frame is received, the node uploads it from the radio frame buffer and checks whether the frame payload is equal to ETB. If so, it executes `+radio-input`. Actually, to switch the input, the word `usart_inject` must be executed to exit the system blocking loop waiting for characters from the USART that in the current AmForth implementation cannot be preempted in other ways.

2.2 Support to the Radio Operations

In order to support the communication among nodes, we defined a number of words to drive the radio of IrisMotes. The low-power AT86RF230 transceiver [15] is connected to the master SPI interface of the microcontroller and to additional control signals, i.e. IRQ and GPIO signals. Essentially, the SPI is used for frame buffer and register access operations, according to the SPI protocol. Although AmForth already provides the words `spi!` and `spi@` for writing and reading a character on the SPI bus, further efforts were needed to configure ports for the specific target device.

We also defined word sets to support the functional specification of the radio device. For instance, in Listing 3 the words `reg_rd` and `reg_wr` specify the operations to be undertaken for reading and writing the radio registers. Similarly, we defined the words `to_framebuf` and `from_framebuf` to upload incoming frames, and download outgoing frames, respectively. Word choices reflect the nomenclature of the radio datasheet.

Uninterruptible code, such as that implementing SPI operations, is enclosed within critical sections. The words `ss_l` and `ss_h` set the SS line of the SPI interface respectively low and high.

Listing 3: Some words of the radio driver

```
: reg_rd ( register_address -- register_value )
      reg_addr_mask and reg_rd_command or
```

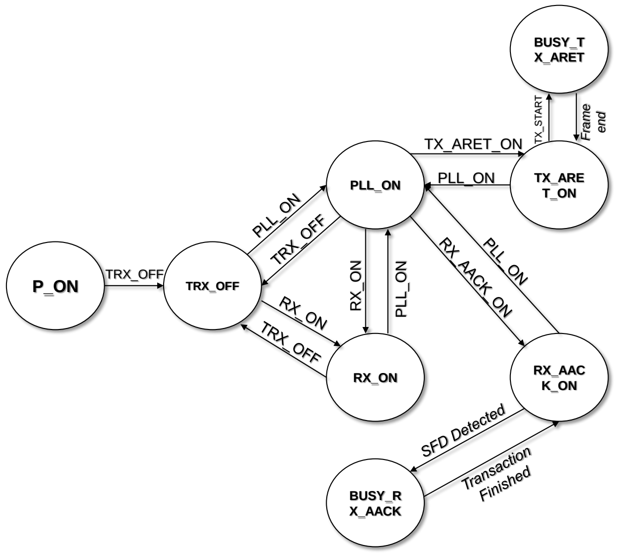

Figure 2: Part of the state diagram representing the set of operating modes of the AT86RF230 according to the datasheet [15]. The label on the arrows represents the command that writes to the TRX_STATUS register causing the transaction to another state. Events are indicated with labels in italics.

```
      critical[
      ss_h ss_l spi! spi@ ss_h
      ]critical
;

: reg_wr ( register_value register_address -- )
      reg_addr_mask and reg_wr_command or
      critical[
      ss_h ss_l spi! spi! ss_h
      ]critical
;

: to_framebuf ( packet_to_send -- packet )
      dup critical[
      ss_h ss_l framebuf_wr_command spi! length spi!
      dup length 0 ?do dup I + 1 + c@ spi! loop
      ss_h ]critical
;

: from_framebuf ( packet -- packet )
      critical[
      ss_h ss_l framebuf_rd_command spi!
      spi@ over c! dup length 0 ?do
      spi@ over I + 1 + c! loop ss_h ]critical
;
```

The radio transceiver operating modes and its transitions can be represented by the state diagram in Figure 2.

To permit a plain alignment between specifications and implementation, the same diagram can be completely ported into Forth definitions as shown in Listing 4, which includes just a restricted number of defined words.

Listing 4: Definitions for radio operating modes and transitions

```
: pll_on ( -- )
      pll_on_state cmd-wr  ;

: trx_off ( -- )
      st-reset ;

: tx_start ( -- )
      tx_start_cmd cmd-wr ;

: rx_on ( -- )
      rx_on_state cmd-wr  ;

: rx_aack_on ( -- )
      pll_on rx_aack_on_state cmd-wr ;

: tx_aret_on ( -- )
```

```
      pll_on tx_aret_on_state cmd-wr ;

: transmit ( packet -- )
     -int IRQ low
      idle? if green led blink else
      trx_off outbound process-tx drop to_framebuf drop
      tx_aret_on tx_start
      then +int
;

: switch-input? ( inbound -- )
      dup payld addr nip c@ $17 = if
      +radio-input $17 usart_inject
      then
;

: received ( inbound -- )
      trx_off
      from_framebuf dup process-rx switch-input?
      payld addr nip swap payld size payld-set
;
```

In particular the words `process-rx` and `process-tx` are deferred words that may be used to process `inbound` and `outbound` buffers. A possible use may be for encryption and decryption purposes. The last three definitions implement frame transmission, input redirection and frame reception. Transmission and reception of frames are signaled by interrupts on the Timer1 Input Capture Trigger. The Interrupt Service Routine in AmForth is also a defined word. Therefore, we defined a word acting as the handler routine and we stored its address as interrupt vector. Our interrupt handler routine reads the IRQ STATUS register and acts as a dispatcher. Since the AT86RF230 differentiates between six interrupt events, it calls the appropriate interrupt handler, according to the interrupt source. For instance, the interrupt generated by either a frame transmission/reception causes the execution of the word `trx_end_isr`. If a correct transmission triggers the interrupt, the radio enters the `rx_aack_on` state, otherwise the frame is downloaded to the `inbound` buffer.

```
: trx_end_isr
     red led blink state?
     tx_aret_on_state = if
     else inbound received
     then trac_status trac !
     rx_aack_on ;
```

2.3 Supporting Sensing and Actuating Tasks

The acquisition of sensory data is the main functionality of WSN nodes. Typically, expansion boards are required to provide the nodes with several sensors simultaneously. As in the case of the radio transceiver driver implementation, we have extended the WSN node dictionary with a number of words to drive sensor boards. Moreover, word sets composed of high level words enable the data sensory acquisition through the different available sensors. For instance, a program to make a node sense the temperature may consist of the single word `temperature` that leaves at the top of the stack the required sensory value. Similarly, the word `luminosity` activates the light sensor, puts the sensory reading atop the stack, and finally disables the sensor. Although the code is concise and expressive, the execution of these words involves low level aspects as reading from the ADC and returning the raw data on the stack. However, we choose high level word names to make the description of a task in natural language and the implementation as similar as possible. The words we defined to support sensing tasks are summarized in Table 1.

WSNs may also include some actuator nodes to change the environmental conditions. An IrisMote can behave as an actuator when connected, for instance, to the MDA300 expansion board that includes two relays, one of which normally opened and the other one normally closed. We defined words to drive the relays and developed a light control application by connecting a LED to the expansion board, as detailed in Section 4.

3 A Forth-based Approach to Enable Symbolic Distributed Processing for WSNs

To implement sophisticated AmI applications, even resource constrained nodes may need to exchange complex information that is not rigidly structured and that may differ from numerical values such as symbolic descriptions and rules. Conventional programming methodologies impose to define in advance the format of the message to

Table 1: Summary table of words used for sensing tasks. Words are indicated together with their “stack effect”

Word name	**Description**
temperature (-- temp_value)	Measure temperature and push the numeric value onto the stack
luminosity (-- light_value)	Measure light and push the numeric value onto the stack
mic (-- mic_value)	Measure sound level and push the numeric value onto the stack
accx (-- accx_value)	Measure the acceleration along the X axis and push the numeric value onto the stack
accy (-- accx_value)	Measure the acceleration along the Y axis and push the numeric value onto the stack
+sounder (--)	Activate the buzzer
-sounder (--)	Disable the buzzer

be exchanged as well as to fix the packet fields where given information must be placed. To overcome this rigidity, interpreters targeting WSN nodes, such as Maté [9], T-RES [6] and TakaTuka [10], have been presented. However, such solutions are based on bytecode transmission and interpretation. Not only the source code expressivity gets lost as the source code is translated into bytecode but also the translation process is, in all effects, a cross-compilation.

In order to retain expressiveness without sacrificing compactness, we let our nodes able to directly exchange and execute Forth code. Indeed, we implemented an abstract mechanism to handle the transmission of code among nodes, and from the terminal shell to nodes. The implementation cost of such an abstraction is quite low in Forth and, at the same time, the support to distributed applications is straightforward, as shown in Listing 5.

Listing 5: Forth words for executable code exchange

```
variable current_pay
variable nest
variable current_buf
variable buf $80 cells allot

: 2dup over over ;
: buf-reset buf current_buf ! ;
: pay-reset outbound payld addr nip current_pay ! ;

: (write) \ i*x addr len  dest_addr -- j*y
 swap cmove
;


: num>str ( number -- string_addr string_len )
      hex 0 <# #s [char] $ hold #> ;

: space+ ( pay_ptr -- pay_ptr+1)
      bl p+
;

: cr+ ( pay_ptr -- pay_ptr+1)
      $0d p+
;

: nest+ ( -- )
      nest @ 1 + nest ! ;
: nest- ( -- )
      nest @ 1 - nest ! ;

: [tell:]? ( addr len -- f )
      s" [tell:]" icompare ;

: [:tell]? ( addr len -- f )
      s" [:tell]" icompare ;

: tell:? ( addr len -- f )
      s" tell:" icompare ;

: :tell? ( addr len -- f )
      s" :tell" icompare ;


: >buf ( addr1 n -- )
      dup >R current_buf @ dup >R (write)
      R> R> + space+ current_buf ! ;

: >pkt ( addr1 n -- )
      dup >R current_pay @ dup >R (write)
      R> R> + space+ current_pay ! ;

: c>pkt ( value -- )
      current_pay @ swap ( c@ -- ) p+ current_pay ! ;

: char>buf ( value -- )
      current_buf @ swap over c! 1 + current_buf !
;


: subst
      0 do buf I + c@ dup ( c@ -- )
      case
      $0e of drop num>str >pkt endof
      $0f of drop num>str >pkt endof
      $10 of drop >pkt endof
      c>pkt
      endcase loop
;

: [endtell] ( flash-addr flash-count -- )
      dup >r buf imove r>
      subst outbound dup current_pay @ cr+
      endpayld transmit
      pay-reset
;

: endtell ( buf buf-len -- )
      nip subst outbound dup current_pay @ cr+
      endpayld transmit
      pay-reset
;

: subst? nest @ 0 = if
         2dup s" ~" icompare if drop drop $0e true else
         2dup s" ~~" icompare if drop drop $0f true else
         2dup s" ~s" icompare if drop drop $10 true else
         drop drop 0 then then then
         else drop drop 0 then
;


: parse-tell ( -- buf buf-len )
   buf-reset
   begin bl word count
        2dup :tell? if
          nest @ 0 > if nest- >buf 0
                  else true
                  then
        else
        2dup tell:? if nest+ >buf 0
        else 2dup [tell:]? if nest @ 3 + nest ! >buf 0
        else 2dup [:tell]? if nest @ 3 - nest ! >buf 0
        else 2dup subst? if char>buf drop drop 0
        else >buf 0
        then then then then then until drop drop
        buf current_buf @ over -
;

: [parse-tell]
   buf-reset
   begin bl word count
        2dup [:tell]? if
          nest @ 0 > if nest @ 3 - nest ! >buf 0
                  else true
                  then
        else
        2dup [tell:]? if nest @ 3 + nest ! >buf 0
        else 2dup tell:? if nest+ >buf 0
        else 2dup :tell? if nest- >buf 0
        else 2dup subst? if char>buf drop drop 0
        else >buf 0
        then then then then then until drop drop
        buf current_buf @ over -
;

: reply ( -- dest_addr)
      inbound src addr @ nip ;
;

: pkt-init 0 nest ! outbound erase
```

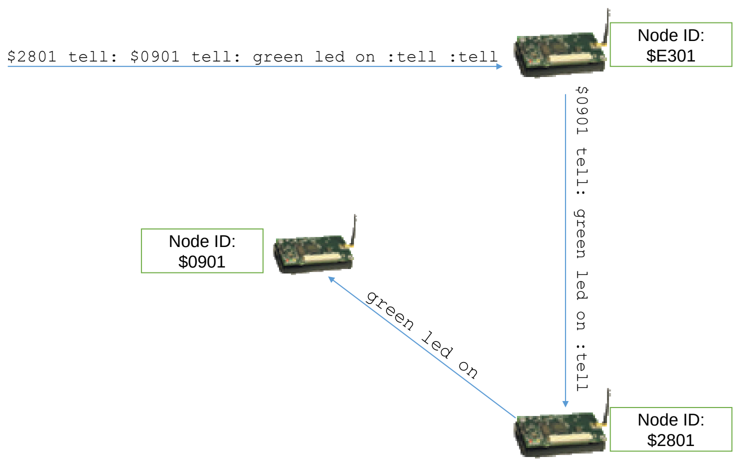
$2801 tell: $0901 tell: green led on :tell :tell
Node ID: $E301
$0901 tell: green led on :tell
Node ID: $0901
green led on
Node ID: $2801

Figure 3: Recursive employment of the `tell: <code> :tell primitive`. Once the node with ID E301 encounters the first `tell:` it parses all the following symbols until the last `:tell` and a frame containing `0901 tell: green led on :tell` is sent to node with ID 2801. The payload interpretation of the payload on the receiving side leads to the sending of a new frame destinated to node 0901 with `green led on` as payload. Once received, node 0901 turn its green LED on.

```
      default-pkt dest addr rot s_addr! drop
      pay-reset
;

: tell:
     pkt-init parse-tell endtell
;


: [tell:]
      postpone pkt-init
      [parse-tell] postpone sliteral
      postpone [endtell]
; immediate
```

Our programming environment and experimental setup is composed of some nodes wirelessly deployed and a wired node that behaves as a bridge to send user inputs to the network.

The syntactic construct for the code exchange among nodes is based on the word `tell:` that parses the input until :`tell` is encountered and sends a default data frame, according to IEEE 802.15.4 standard, to the node holding the MAC address placed on top of the stack.

To tell all the nodes in the radio range to turn their green LED on, a simple line of code is all that it needs to be typed on the bridge node shell:

```
bcst tell: green led on :tell
```

As a consequence, the microcontroller on the bridge node interprets the text typed by the user and creates a default data frame with the broadcast address as destination, containing the program to be sent, `green led on`, as payload.

A recursive usage of code exchange, through nested `tell:` <*code*> `:tell` constructs, permits commands to hop form one device to another before reaching the final destination, as shown in Figure 3. From a mere semantic standpoint, the sense is *"to tell a node to tell another node to do something"*.

The code to be remotely executed may contain syntactic placeholders that are substituted at runtime with the content of the top of the stack using a hexadecimal representation. For the sake of clarity, our implementation consists in a two pass parsing process. An intermediate substitution of such special markers takes place in the first pass, while the items on top of the stack definitively replaces placeholders during the second pass.

Such special markers are:

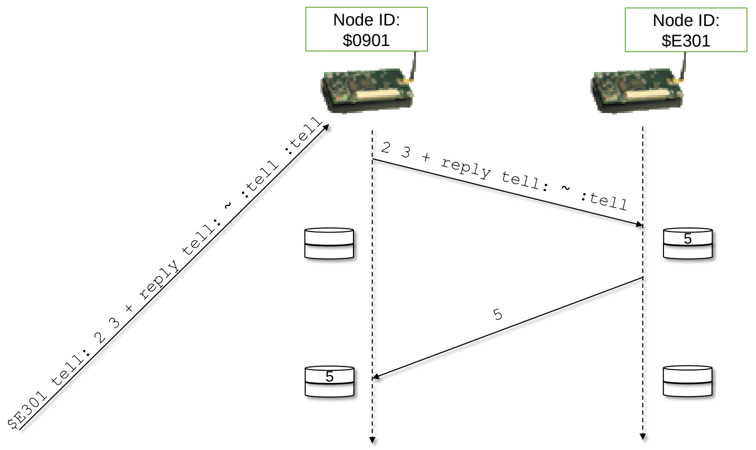
Node ID: $0901
Node ID: $E301
$E301 tell: 2 3 + reply tell: ~ :tell :tell
2 3 + reply tell: ~ :tell
5
5
5

Figure 4: The node with address $0901 interprets the code it receives and tells the node with address $E301 to perform the sum between 2 and 3, and then to reply with the value on top of its stack. Even though the reply message consists only in a literal value, it is interpretable Forth code and it is simply executed by node $0901 leaving 5 on top of its stack.

- $\sim$ for a single cell value
- $\sim \sim$ for a double cell value
- $\sim$s for strings

Instead of implementing state-smart words for code exchange [16], we defined the compile-time construct `[tell:]`<*code*>`[:tell]`.

An example of code exchange between two nodes is described in Figure 4. Incorporating such high level abstraction on resource constrained devices leaves plenty of room for the development of WSN applications that natively support distributed processing. The word sets composing our software platform are reported in Figure 5 along with their size in terms of number of words and Flash memory occupation.

Besides providing an on-board interpreter that does not need cross-compilation, our approach compares favorably with the aforementioned interpreter-based architectures with respect to memory usage, as we assessed with tests in our experimental setup. Where possible, as for TakaTuka and Maté, we compiled the software platforms for the IrisMote or for the quite similar MicaZ hardware. T-RES, instead, only runs on the WiSMote hardware platform.

Results, reported in Figure 6, confirm that the implementation of the interpreter above a general purpose operating system occupies much of the available memory, as in the case of Maté and T-RES. As scripts are stored in RAM, not enough space is left for the development of complex applications, even in the presence of the double-sized RAM of WiSMotes. Our Forth-based approach, instead, compactly keeps application code in the relatively abundant Flash, while RAM just holds temporary data as variable values, buffers and stacks.

More in detail, the memory footprint of all the platform word sets is 5170 bytes of Flash, as reported in Figure 6, and 1026 bytes of RAM. Including the underlying AmForth, the overall footprint of our platform is 18693 bytes of Flash memory and 1321 bytes of RAM memory.

4 Application Development on WSNs

We have developed different applications for WSNs to test both our approach and our software platform. As a first step, we designed and implemented a working telnet-like remote shell on the bridge node to be actually used as a development tool [17]. Using the remote shell application on the bridge node through a serial terminal, the programmer can interact with a remote node that is reachable by the bridge node.

Besides debugging and node reprogramming, this application can serve different purposes such as the inspection of the state of a remote node or the acquisition of sensory readings as if the remote node were physically connected through the serial line.

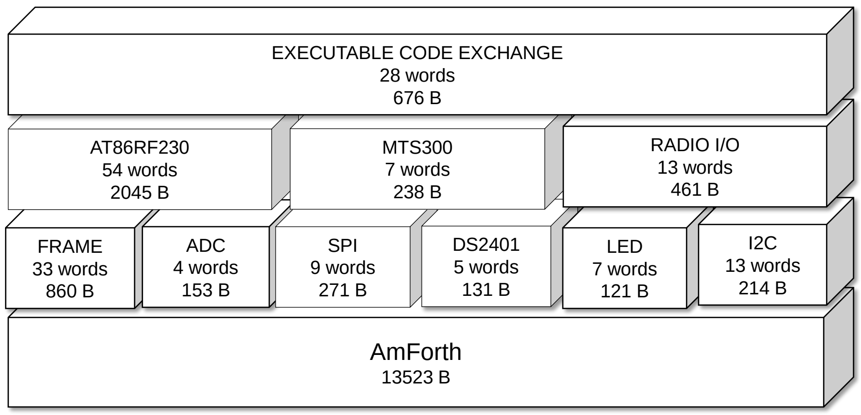

Figure 5: A comprehensive view of the main word sets we defined and that compose our software platform. For each word set, the number of words and the Flash usage in bytes are indicated.

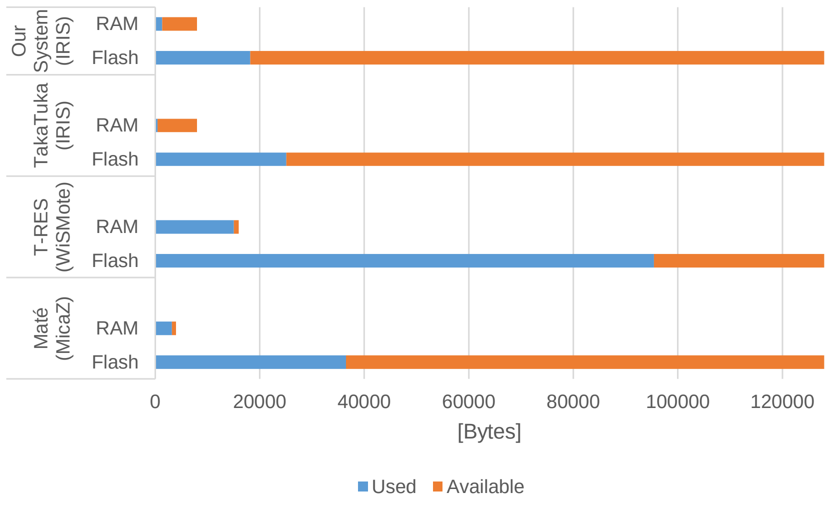

Figure 6: Memory footprint of our software platform along with some representative interpreter-based architectures.

The code is fully functional, and has been extensively used in our experimentations. We defined a number of words to redirect the output to the outgoing message, to display incoming messages from the inspected node, and to implement the remote shell loop. The resulting implementation is quite readable and understandable. The almost complete remote shell application code is shown in Listing 6. Although few additional words are omitted, their description can be found in Table 2.

Listing 6: Code for a simple remote shell application

```
80 constant cmd-maxlen
variable cmd cmd-maxlen cells allot
variable cmd-len
variable node_id
variable timeout

: input-send ( -- )
    cmd cmd-len @
    node_id @ [tell:] ~s [:tell] ;

: rshell-task (  -- )
    payld-reset
    input-send
    timeout @ wait-answer if
    payld-print then ;
```

```
: user-input ( -- )
    cmd cmd-maxlen accept ( -- len )
    cmd-len ! ;

: close ( -- )
    node_id @ [tell:] -radio-output
    [:tell] quit ;

: rshell-loop ( -- )
    begin
     cr ." rsh>" user-input
     close?
     if close
     else rshell-task
     then
    again ;

: on-timeout ( -- )
    ." Connection timeout." cr ;

: welcome-msg
    ." Welcome to the remote shell
 application!" cr
    ." Enter 'close' to close the
 application" cr ;

: rshell ( id -- )
    welcome-msg
    2000 timeout  !
    dup node_id !
    [tell:] +radio-output [:tell]
    rshell-loop  ;
```

Table 2: Summary table of additional words used in the remote shell application

Word (before -- after)	**Description**
+radio-output (--)	Redirect the output to the radio. This word is part of the Radio I/O word set
-radio-output (--)	Redirect the output to the UART. This word is part of the Radio I/O word set
radio-input? (timeout -- flag)	Check for incoming radio messages. If no message arrives before `timeout` milliseconds leave false on the stack, otherwise leave true
payld-reset (--)	Set to 0 the incoming payload length and its current pointer
payld-print (--)	Display the incoming frame content (i.e. the payload)
wait-answer (timeout -- flag)	Wait for incoming radio frame for a predefined period of time specified by the `timeout` variable. If the timeout expires without receiving any answer message, an exception handled by `on-timeout` occurs
user-input (--)	Wait for user input and store its content in the `cmd` buffer and its length in the `cmd-len` variable for further processing by `close?` and `input-send`

A desirable use of WSN nodes is monitoring the environment to react to undesired events. A role-based working implementation differentiates the words defined on remote nodes on the basis of their role in the network.

For instance, the actuator node dictionary could include words to trigger an alarm if the luminosity value exceeds a predefined threshold. A node provided with a light sensor may regularly perform the luminosity measurement and tell its neighbor to forward it to the actuator.

Such words explicitly make use of the syntactic construct for distributed code exchange. However, the designer may interactively set the topology, the threshold, the actuator node and may start the event detection application by interacting with the bridge node shell.

The code to implement such an application is provided in Listing 7 and consists in few words defined on the three kinds of nodes. With respect to the baseline of our platform, the increment of RAM usage on the sensor node for the application is just 6 bytes, while additional 156 Flash bytes are required to store the word definitions for timer3 management and the application. Since the routines for timer3 are not needed on the forwarder and actuator nodes, Flash and RAM increments for both are just 86 and 4 bytes respectively.

More importantly, the overall Flash and RAM memory footprint of the application and the software platform, even considering 21 additional bytes for the turnkey definitions, is 18714 bytes of Flash memory and 1321 bytes of RAM memory. Such result is even lower than the baselines of the other platforms –that is without any application– as can be deduced from Figure 6.

Listing 7: Distributed event detection application on WSN nodes

```
\ Defined on the sensor node
variable neighbor
variable threshold

: luminosity-check
        luminosity
        threshold @ > if
        neighbor @ [tell:] alarm [:tell]
        then ;

: light-monitoring
        ['] luminosity-check
        5seconds timer3.init timer3.start ;

\ Defined on the forwarder node
\ and on the actuator node

variable actuator
variable neighbor

: same ( -- outbound)
        outbound inbound over length copy ;

: message ( dest_addr outbound --outbound)
         dest addr rot s_addr! ;

: propagate
        transmit ;

: actuator?
        actuator @ 1 = ;

: alarm
        actuator? not if
        neighbor @ same message propagate
        else +sounder 1000 ms -sounder
        then ;
```

Furthermore, instead of an alarm, the actuator may directly switch the light off once the luminosity exceeds the threshold value through a redefinition of the word `alarm` (Listing 8).

Listing 8: Redefinition on the actuator node of the word that triggers the alarm

```
: alarm
        actuator? not if
        neighbor @ same message propagate
        else     light off then ;
```

In previous work [18] we have also showed how to support smart applications that exploit symbolic reasoning. We enriched a Forth formalism for Fuzzy Logic by VanNorman [19] with the possibility to exchange definitions and evaluations among nodes. Instead of reasoning about crisp values, resource constrained nodes process the fuzzy variables `temp` and `lightexp` that can be easily defined on deployed nodes. Further words such as `fvar` define the related membership functions. By exploiting executable code exchange, a fuzzy variable definition can be easily distributed among nodes even after their deployment. After defining the membership functions `lightexp.low`, `lightexp.medium`, `lightexp.high` and `temp.low`, `temp.medium`, `temp.high`, the following code makes a node measure and fuzzify light exposure:

```
lightexp measure apply
```

while the code:

```
lightexp.low @
```

pushes onto the stack the truth value resulting from the fuzzification phase. For instance, rather than through a thresholding process, a device can establish if it is close to the window through the evaluation of fuzzy rules in the form:

```
temp.high @ lightexp.high @ &
=> close-to-window
```

A node can request the others to update their fuzzy temperature values as follows:

```
bcst temp fvar-remote-update
```

The word `fvar-remote-update` evaluates `temp` and broadcasts a message containing the Forth code to update the three membership values. The frame payload includes a repetition of the structure:

```
<truth> <membership func> fvar-update
```

for each membership function of the argument fuzzy variable. When a node receives the message, it interprets the command updating the truth values of its local membership functions. The combination of symbolic reasoning with executable code exchange makes even resource constrained devices able to process and exchange qualitative information about the physical phenomenon.

5 Conclusions

As remarked in literature, common programming methodologies for WSNs lack proper programming abstractions for the development of distributed applications. The standard practice consists in linking the application, written in C-derived programming languages, with a general-purpose operating system at the end of a cross-compilation process. All this proves rigid and time consuming. To overcome these limitations, the adoption of interpreters for high-level languages to be run on established operating systems has been proposed. Nevertheless, existing approaches consist in several software layer implementations that collide with the resource constraints of nodes.

In this paper, we detailed the implementation of an alternative Forth-based approach that implements a minimal but extensible operating system featuring common WSN functionalities along with symbolic distributed processing through executable code exchange. The Forth-based software platform we have implemented is quite compact. Indeed, we showed how a symbolic distributed AmI event detection application can be implemented with a total memory usage that is less than the mere baselines of relevant interpreter-based software platform for WSNs. In further experimentations we will compare our Forth environment to other existing interpreter-based platforms for WSNs in terms of efficiency, interpretation overhead and energy consumption.

References

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

[2] EEPK Gilbert, K Baskaran, and EB Elijah Blessing. Research Issues in Wireless Sensor Network Applications: A Survey. *International Journal of Information and Electronics Engineering*, 2(5):702–706, 2012.

[3] Muhammad Omer Farooq and Thomas Kunz. Operating systems for wireless sensor networks: A survey. *Sensors*, 11(6):5900–5930, 2011.

[4] Adi Mallikarjuna V. Reddy, A.V.U. Phani Kumar, D. Janakiram, and G. Ashok Kumar. Wireless sensor network operating systems: a survey. *Int. J. Sen. Netw.*, 5(4):236–255, August 2009.

[5] Luís M Oliveira and Joel J Rodrigues. Wireless Sensor Networks: a Survey on Environmental Monitoring. *Journal of communications*, 6(2):143–151, 2011.

[6] D. Alessandrelli, M. Petracca, and P. Pagano. T-Res: Enabling Reconfigurable In-network Processing in IoT-based WSNs. In *Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on*, pages 337–344, May 2013.

[7] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011.

[8] L. Evers, P.J.M. Havinga, J. Kuper, M.E.M. Lijding, and N. Meratnia. SensorScheme: Supply chain management automation using Wireless Sensor Networks. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 448–455, Sept 2007.

[9] Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ACM Sigplan Notices*, volume 37, pages 85–95. ACM, 2002.

[10] Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rhrup, and ZastashA. Uzmi. Optimized Java Binary and Virtual Machine for Tiny Motes. In Rajmohan Rajaraman, Thomas Moscibroda, Adam Dunkels, and Anna Scaglione, editors, *Distributed Computing in Sensor Systems*, volume 6131 of *Lecture Notes in Computer Science*, pages 15–30. Springer Berlin Heidelberg, 2010.

[11] W. Munawar, M.H. Alizai, O. Landsiedel, and K. Wehrle. Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–6, May 2010.

[12] BrianH. Watts. FORTH, a Software Solution to Real-time Computing Problems. *Behavior Research Methods, Instruments, & Computers*, 18(2):228–235, 1986.

[13] Iris Datasheet, 2013. Available online at `http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf`.

[14] Amforth documentation, 2013. Available online at `http://amforth.sourceforge.net/amforth.pdf`.

[15] At86rf230 datasheet, 2013. Available online at `http://www.atmel.com/images/doc5131.pdf`.

[16] M Anton Ertl. State-smartness— Why it is Evil and How to Exorcise it. *EuroForth98*, 1998.

[17] Salvatore Gaglio, Giuseppe Lo Re, Gloria Martorella, and Daniele Peri. A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–8, Sept 2014.

[18] Salvatore Gaglio, Giuseppe Lo Re, Gloria Martorella, and Daniele Peri. High-level Programming and Symbolic Reasoning on IoT Resource Constrained Devices. In *Accepted at The First International Conference on Cognitive Internet of Things (COIOTE 2014)*, October 2014.

[19] R. VanNorman. Fuzzy Forth. *Forth Dimensions*, 18:6–13, March 1997.