



**AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE**

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,  
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa magisterska

*Realizacja podstawowej funkcjonalności maszyny wirtualnej Erlanga  
dla systemu FreeRTOS*

*Implementation of basic features of Erlang Virtual Machine for  
FreeRTOS*

Autor: *Rafał Studnicki*  
Kierunek studiów: *Informatyka*  
Opiekun pracy: *dr inż. Piotr Matyasik*

Kraków, 2014

*Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.*



## Spis treści

<b>1. Przykładowe aplikacje</b> .....	7
1.1. Silnia.....	7
1.1.1. Cel aplikacji .....	7
1.1.2. Uzyskane wyniki.....	8
1.1.3. Wnioski .....	9
1.2. Kontrola diod LED przez procesy .....	11
1.2.1. Cel aplikacji .....	11
1.2.2. Uzyskane wyniki.....	13
1.2.3. Wnioski .....	14
1.3. Sterownik RFM73 .....	14
1.3.1. Cel aplikacji .....	14
1.3.2. Uzyskane wyniki.....	17
1.3.3. Wnioski .....	17
<b>Bibliografia</b> .....	19



# 1. Przykładowe aplikacje

W niniejszym rozdziale pracy zaprezentowano przykładowe aplikacje zaimplementowane w języku Erlang i uruchomione na maszynie wirtualnej opisywanej w pracy. Platformą, na której uruchamiane były przykłady był mikrokontroler LPC1769 z procesorem ARM Cortex-M3 pod kontrolą mikrojądra FreeRTOS.

Wszystkie aplikacje zostały skompilowane przy użyciu narzędzia opisanego w dodatku ???. Kody źródłowe aplikacji zostały umieszczone na płycie CD dołączonej do pracy.

## 1.1. Silnia

### 1.1.1. Cel aplikacji

Aplikacja miała na celu uruchomienie przykładowego sekwencyjnego programu zaimplementowanego w języku Erlang w dwóch wersjach: z rekurencją ogonową oraz bez tego typu rekurencji.

Rekurencja ogonowa charakteryzuje się tym, że wynik wywołania funkcji rekurencyjnej całkowicie zależy od wyniku rekurencyjnego wywołania funkcji. Nie ma zatem konieczności powrotu do poprzedniego wywołania funkcji w celu ustalenia aktualnej wartości funkcji, a co za tym idzie zapisywania na stosie adresu powrotu i wyrażeń potrzebnych do wyliczenia zwracanego wyniku. Kompilator Erlanga automatycznie wykrywa funkcje ogonowo-rekurencyjne i optymalizuje kod pośredni pod kątem użytego rozmiaru stosu (por. np. opis instrukcji `call` i `call_only` na liście instrukcji kodu pośredniego na str. ??).

Listingi 1.1 oraz 1.2 przedstawiają kody źródłowe dwóch funkcjonalnie równoważnych sobie funkcji (`fac/1`) obliczających silnię liczby naturalnej.

Funkcja z modułu `fac` wykorzystuje do tego celu tradycyjną rekurencję, uzależniając wynik zwrócony przez funkcję nie tylko od rekurencyjnego wywołania samej siebie, ale także od argumentu z jakim została wywołana.

Funkcja zaimplementowana w module `fac2` wykorzystuje dodatkowy argument, tzw. akumulator, którego aktualna wartość przekazywana jest wraz z każdym kolejnym wywołaniem, a na samym końcu zwracana. Dzięki zastosowaniu tego podejścia funkcja ta jest ogonowo-rekurencyjna, co pozwala na optymalizację użycia stosu przez kompilator Erlanga w kodzie pośrednim.

```
1 -module(fac) .
2
3 -export([fac/1]) .
4
5 fac(0) ->
6     1;
7 fac(N) ->
8     N*fac(N-1) .
9
10
11
```

Listing 1.1: Kod modułu `fac.erl`

```
1 -module(fac2) .
2
3 -export([fac/1]) .
4
5 fac(N) ->
6     fac(N, 1) .
7
8 fac(0, Acc) ->
9     Acc;
10 fac(N, Acc) ->
11     fac(N-1, N*Acc) .
```

Listing 1.2: Kod modułu `fac2.erl`

Celem eksperymentu było uruchomienie dwóch ww. funkcji obliczających silnię dla różnych wejściowych liczb naturalnych od 1 do 200. Pozwoliło to na sprawdzenie działania podstawowych elementów maszyny wirtualnej, m.in. interpretera kodu pośredniego, *garbage collector*a czy arytmetyki dużych liczb (do zapisania wyniku 200! potrzebnych jest 156 bajtów).

### 1.1.2. Uzyskane wyniki

Silnia została obliczona dla wejściowych liczb: 1, 11 (wynik 11! jest ostatnim, jaki mieści się na 28 bitach i może zostać przechowany jako wyrażenie typu **SMALL**), 12 (wynik 12! jest ostatnim, jaki mieści się na 32 bitach i jest zarazem pierwszym, który do przechowania potrzebuje typu danych **BIGNUM**), 25, 50, 75, 100, 125, 150, 175 i 200. Dla wszystkich wartości wejściowych, dla obu modułów, uzyskano poprawne wartości silni.

Obliczenia zostały wykonane w kontekście procesu Erlanga, który był jedynym uruchomionym w tym momencie na maszynie wirtualnej.

Czas wykonania mierzony był przy użyciu sprzętowego licznika wbudowanego w mikrokontroler, taktowanego zegarem o częstotliwości 1 MHz.

Rezultaty uzyskane w trakcie uruchomień zostały zaprezentowane na rysunku 1.1.

Zgodnie z oczekiwaniami rozmiar sterty procesu w przypadku modułu `fac` rósł znacznie szybciej niż w przypadku modułu `fac2`, osiągając aż 610 słów maszynowych w porównaniu do 90 słów w przypadku funkcji ogonowo-rekurencyjnej dla obliczenia wyniku 200!. Na samo przechowanie stosu wywołań w pierwszym przypadku konieczne jest 400 słów maszynowych (200 liczb **SMALL** i 200 adresów powrotu).

Dużo większy rozmiar sterty dla pierwszego z modułów miał bezpośredni wpływ na dużo rzadsze uruchomienia *garbage collector*a. Ponieważ wyrażenia zajmujące znaczną część sterty (duże liczby) potrzebne były tylko w czasie jednego rekurencyjnego wywołania funkcji, przy każdym jego uruchomieniu



zwalniana była dużą część pamięci. Większa część zaalokowanej sterty mogła zostać zatem później wykorzystana przez kolejne wywołania funkcji.

Z tego samego powodu w przypadku funkcji ogonowo-rekurencyjnej, *garbage collectorowi* udało zwolnić się większą część pamięci (ok. 4500 słów maszynowych w porównaniu do ok. 3500 słów maszynowych).

Sam czas wykonywania kodu był nieco większy w przypadku modułu `fac` (2981  $\mu$ s w porównaniu do 2108  $\mu$ s), co wynika bezpośrednio z większej liczby instrukcji w kodzie pośrednim. Czas spędzony na odśmiecaniu sterty procesu (ok. 43% łącznego czasu wykonania programu w przypadku modułu `fac2` w porównaniu do ok. 21% czasu dla drugiego modułu) miał jednak decydujący wpływ na łączny czas obliczenia silni, który okazał się wyraźnie większy dla modułu z funkcją ogonowo-rekurencyjną (4920  $\mu$ s w porównaniu do 3750  $\mu$ s).

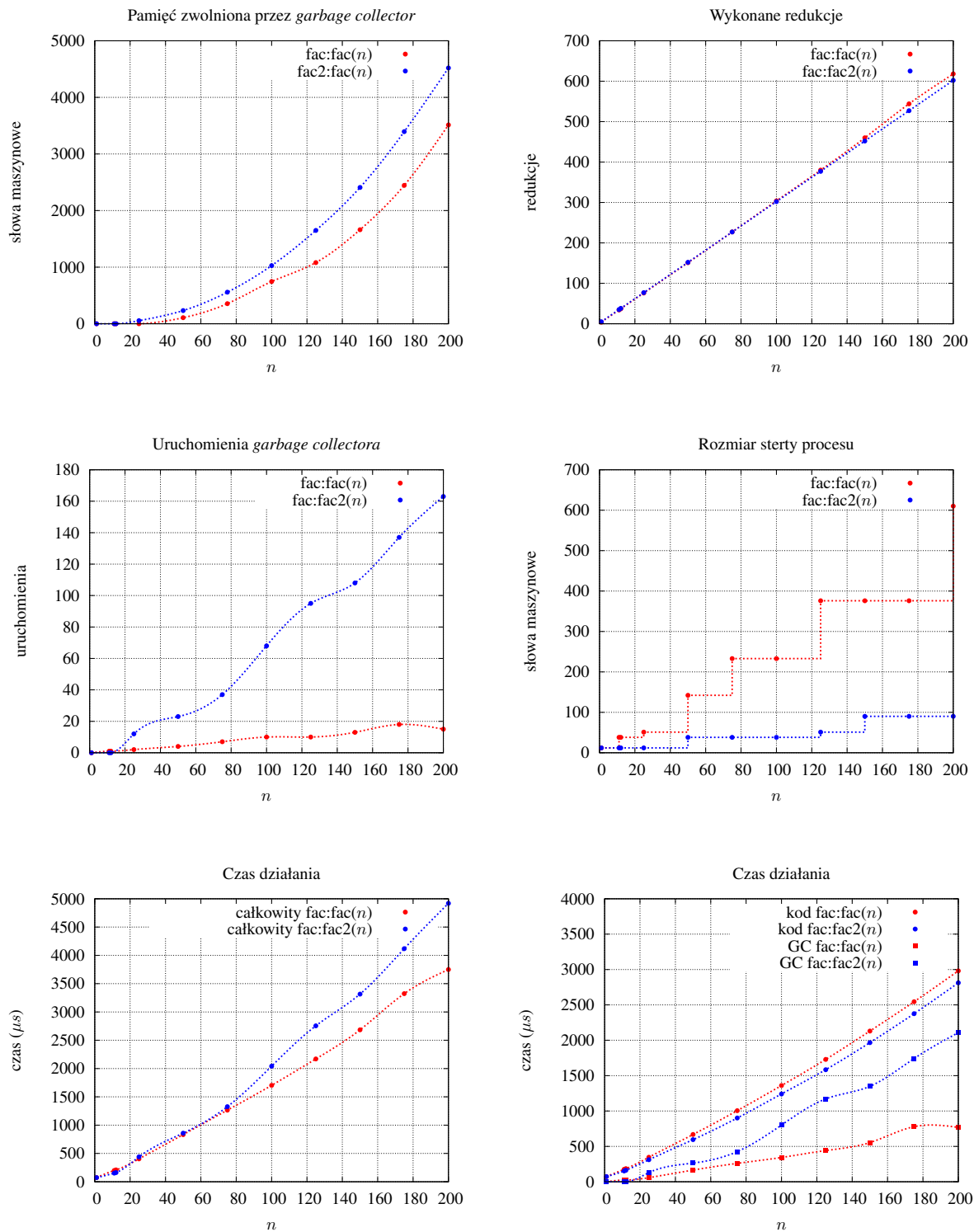
Dla porównania, obie powyższe funkcje obliczające silnię z 200 na maszynie wirtualnej BEAM na komputerze z systemem operacyjnym MacOS X i procesorem Intel Core i7, wykonały się w ok. 1800  $\mu$ s. Należy wspomnieć, że początkowy rozmiar sterty procesu został ustawiony na 12 słów maszynowych, tak jak jest to w przypadku maszyny implementowanej w pracy.

Różnicę można również zauważyć w liczbie wykonanych redukcji przez procesy, która jest nieco większa w przypadku wykonywania kodu z modułu `fac`. Powodem tego jest fakt, że w całkowitą liczbę redukcji procesu wliczany jest czas uruchomienia *garbage collector*a, ale tylko w przypadku gdy jest on uruchamiany przez instrukcje z rodziny `allocate` (opkody 12-15). Jeżeli *garbage collector* uruchomiony zostanie w trakcie operacji arytmetycznej, redukcje nie są doliczane. Ponieważ kod modułu `fac2` nie używa stosu, liczba redukcji przez niego wykonana pochodzi tylko i wyłącznie z wywołań funkcji.

### 1.1.3. Wnioski

Eksperymentalne uruchomienia funkcji obliczających silnię zwróciły poprawne wyniki, testując w ten sposób arytmetykę dużych liczb zaimplementowaną w maszynie wirtualnej. Zmierzone czasy wykonania zarówno kodu jak i odśmiecania mogą stanowić punkt odniesienia przy przewidywaniu narzutu, jaki do aplikacji uruchamianej na mikrokontrolerze może wprowadzić maszyna wirtualna. Na ich podstawie można wyciągnąć wniosek, że rekurencja ogonowa co prawda wykorzystuje zdecydowanie mniej pamięci, jednak przez dużą liczbę odśmieceń zajmuje więcej czasu.

W momencie projektowania funkcji w Erlangu, która uruchamiana będzie na szybkim procesorze i przy dostępnej dużej ilości pamięci, wybór pomiędzy rekurencją ogonową a tradycyjną nie ma większego znaczenia (o ile liczba wywołań nie jest bardzo duża, mogąca spowodować skończenie się dostępnej pamięci) i wybór implementacji powinien zostać podyktowany czytelnością kodu funkcji. Jednak w przypadku urządzeń wbudowanych, ze względu na bardzo ograniczone rozmiary zasobów, zawsze powinna być wybierana rekurencja ogonowa, która po optymalizacji kompilatora nie będzie używać stosu procesu w momencie rekurencyjnych wywołań funkcji. Wydajność konkretnej aplikacji, poprzez zmniejszenie liczby uruchomień *garbage collector*a, może zostać poprawiona przez wybór w pliku konfiguracyjnym początkowego rozmiaru sterty procesu, adekwatnego do jego logiki.



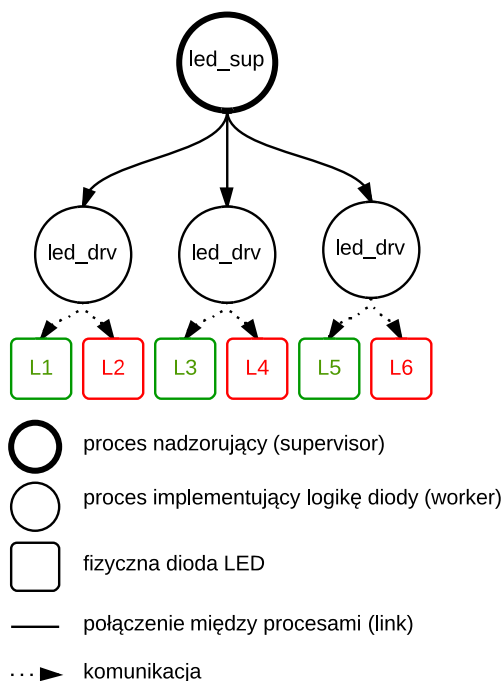
Rysunek 1.1: Porównanie wyników uruchomienia modułów `fac` oraz `fac2` na implementowanej maszynie wirtualnej.

## 1.2. Kontrola diod LED przez procesy

### 1.2.1. Cel aplikacji

Celem aplikacji było uruchomienie przykładowego programu korzystającego ze współbieżnych cech języka Erlang na zaimplementowanej maszynie wirtualnej. Aplikacja miała pozwolić na sprawdzenie działania takich funkcjonalności maszyny wirtualnej jak: przesyłania i odbierania wiadomości między procesami, propagacji zakończenia działania procesu (*link*), mechanizmów zarządzania czasem (przetwarzania i wysyłania wiadomości do innych procesów z opóźnieniem) czy stabilnością działania automatycznego zarządzania pamięcią.

W tym celu zaimplementowano dwa moduły: `led_sup` i `led_drv`, których zadaniem była kontrola diod LED, przez funkcje wbudowane kontrolujące GPIO, w dwóch kolorach: czerwonym i zielonym. W systemie uruchomiony był jeden proces implementujący logikę pierwszego z modułów oraz trzy procesy — drugiego z nich. Każdy z procesów `led_drv` kontrolował stan jednej zielonej diody — zapalał lub gasił ją po otrzymaniu wiadomości z procesu `led_sup`. Proces ten, w odstępie półsekundowym, wysyłał do losowego procesu wiadomość o zmianie stanu diody. Dodatkowo, w logice procesu ustawione zostało przeterminowanie, efektem którego było zliczanie sytuacji, gdy żadna wiadomość kontrolująca stan diody nie przyszła do skrzynki odbiorczej procesu w odstępie półtorej sekundy. Efektem wystąpienia takiego zdarzenia pięty raz w czasie życia procesu skutkowało jego zakończeniem z błędem.

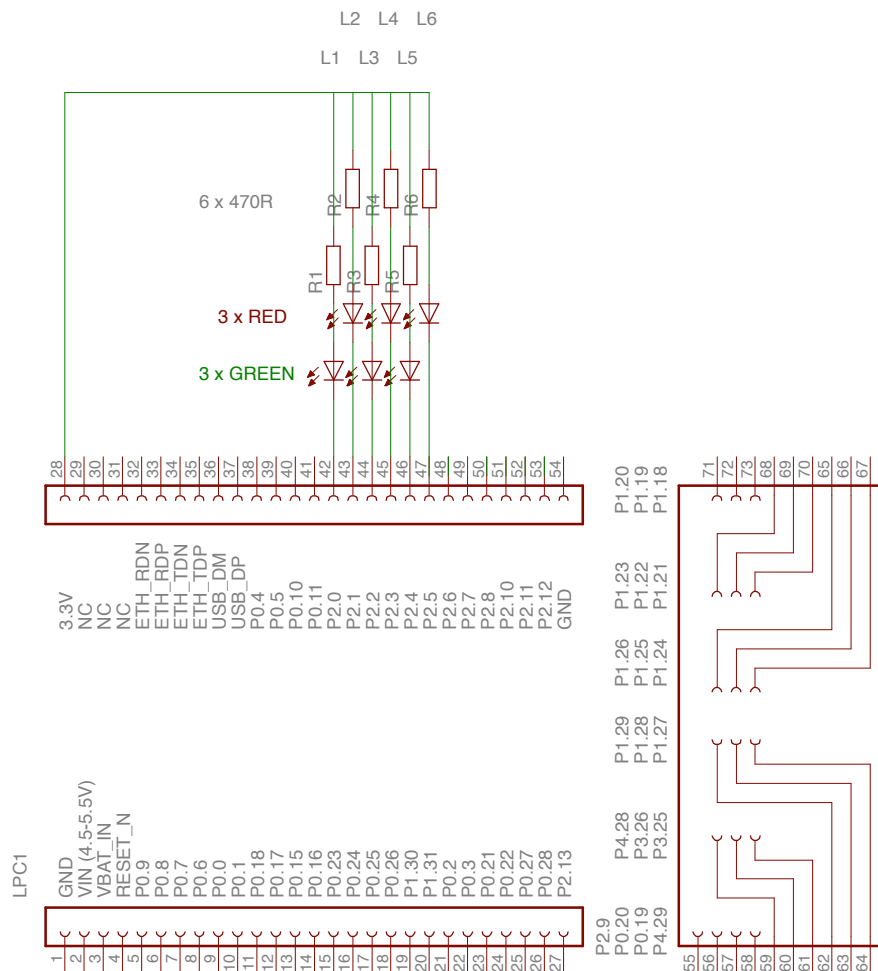


Rysunek 1.2: Struktura procesów w aplikacji i kontrolowanych przez nie diod

Stan systemu wizualizowały dodatkowo trzy czerwone diody, które były zgaszone w momencie gdy

procesy kontrolujące odpowiadające im diody zielone były uruchomione i zapalone w przeciwnym wypadku. Pojedynczy proces był procesem nadzorczym (*supervisor*), którego zadaniem było ponowne uruchamianie kończących się procesów kontrolujących diody zielone. W tym celu, proces ten musiał działać jako proces przechwytyjący wyjścia procesów z nim powiązanych jako wiadomości (co w języku Erlang realizowane jest poprzez ustawienie flagi procesu `trap_exit` na `true`). Dla czytelniejszej prezentacji stanu systemu w przykładzie, proces kontrolujący diodę zieloną restartowany był dopiero po upływie przynajmniej sekundy po otrzymaniu wiadomości o zakończeniu się jego poprzednika.

Zależności między procesami uruchomionymi w ramach przykładowej aplikacji zostały zaprezentowane na rysunku 1.2. Fizyczna realizacja podłączeń diod, sterowanych stanem niskim, do mikrokontrolera wykorzystanego w przykładzie przedstawiona została na rysunku 1.3. Na obu rysunkach diody zostały oznaczone tymi samymi symbolami.



### 1.2.2. Uzyskane wyniki

Kod aplikacji, poza wspomnianymi modułami, składał się z częściowo zaimplementowanych modułów `lists` oraz `random` zawierających funkcje pomocnicze w realizacji implementacji. Należały do nich takie funkcje jak np. `lists:keyfind/3` czy `lists:delete/2`.

Tabela 1.1 prezentuje poziom zajętości pamięci dostępnej w mikrokontrolerze przez poszczególne elementy uruchomionej maszyny wirtualnej oraz załadowanych pięciu modułów składających się na całą aplikację.

W trakcie testowych uruchomień systemu problemem okazało się dynamiczne tworzenie i usuwanie zadań systemu FreeRTOS. Każda operacja tego typu wymagała alokacji nowego stosu dla zadania, który, jak udało się ustalić doświadczalnie, do prawidłowego działania powinien składać się z przynajmniej 350 słów maszynowych. Częste zwalnianie pamięci procesów, które kończą swoje działanie i alokacja pamięci dla nowotworzonych procesów prowadziły do fragmentacji pamięci. W efekcie, np. pomimo dostępnych ponad 5 kB pamięci RAM nie można było zaalokować obszaru pamięci dla stosu przeznaczonego dla nowego procesu. Rozwiązaniem tego problemu okazało się utworzenie, zaalokowanej na samym początku działania maszyny wirtualnej, puli zadań systemu FreeRTOS do których przypisywane były procesy tworzone w trakcie działania systemu. Wobec powyższego, w trakcie uruchomienia aplikacji jedynymi dynamicznie zarządzanymi obszarami pamięci były sterty procesów i wiadomości przesyłane między procesami, będące na tyle niewielkie w stosunku do dostępnej pamięci, że nie powodowały fragmentacji pamięci.

Element systemu	Zajęta pamięć [B]
Tablica atomów (maks. 100 pozycji)	3784
Tablica eksportów (maks. 100 pozycji)	440
Funkcje wbudowane	1040
Moduły, w tym:	11000
<code>led_drv</code>	2160
<code>led_sup</code>	5944
<code>lists</code>	1656
<code>main</code>	560
<code>random</code>	680
Pula procesów (4 procesy)	6584
<b>Suma</b>	<b>22848</b>

Tablica 1.1: Pamięć zajęta przez poszczególne uruchomione elementy maszyny wirtualnej

W celu sprawdzenia stabilności działania maszyny wirtualnej pod kątem realizacji przeterminowań a także ewentualnych wycieków pamięci powodowanych przez *garbage collector* przykładową aplikację pozostawiono uruchomioną na 12 godzin. Następnie zestawiono zestaw statystyk zbieranych przez

uruchomioną maszynę wirtualną co minutę w czasie jej uruchomienia. Statystyki te zaprezentowano w formie wykresów na rysunku 1.4.

Różna liczba uruchomionych procesów w czasie zależna jest od liczby procesów oczekujących na ponowne uruchomienie w momencie dokonania odczytu statystyki. Co za tym idzie, wahania widoczne są również na wykresie przedstawiającym rozmiar pamięci zajmowanej przez stertry wszystkich uruchomionych procesów oraz rozmiar wolnej pamięci RAM w systemie. Niemniej, wskazania wszystkich wymienionych w czasie są stabilne (znajdują się w tym samym zakresie przez cały czas działania aplikacji). Z kolei takie wartości jak: liczba uruchomień *garbage collector*a, rozmiar pamięci odzyskanej przez niego, sumaryczna liczba wysłanych wiadomości przez procesy czy sumaryczna liczba wystartowanych procesów w systemie przez cały czas działania aplikacji rosły liniowo. Świadczy to o stabilnej i poprawnej obsłudze aplikacji przez maszynę wirtualną wraz z upływem czasu.

### 1.2.3. Wnioski

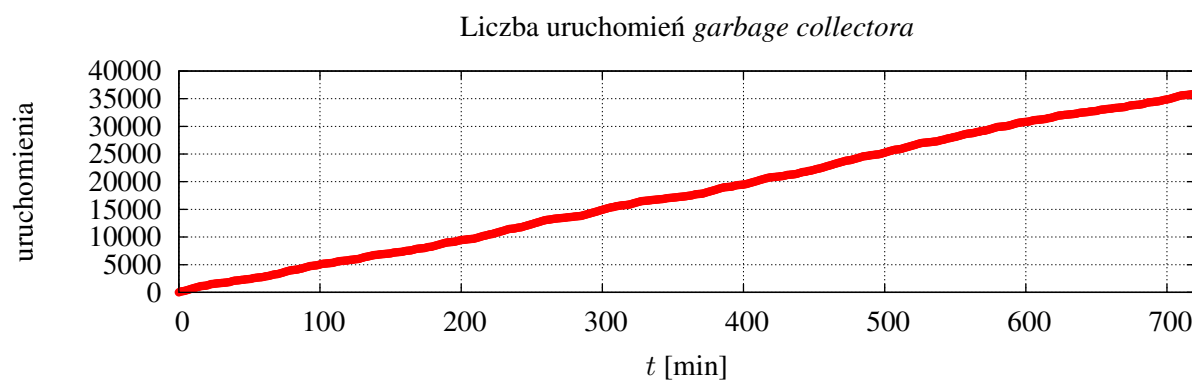
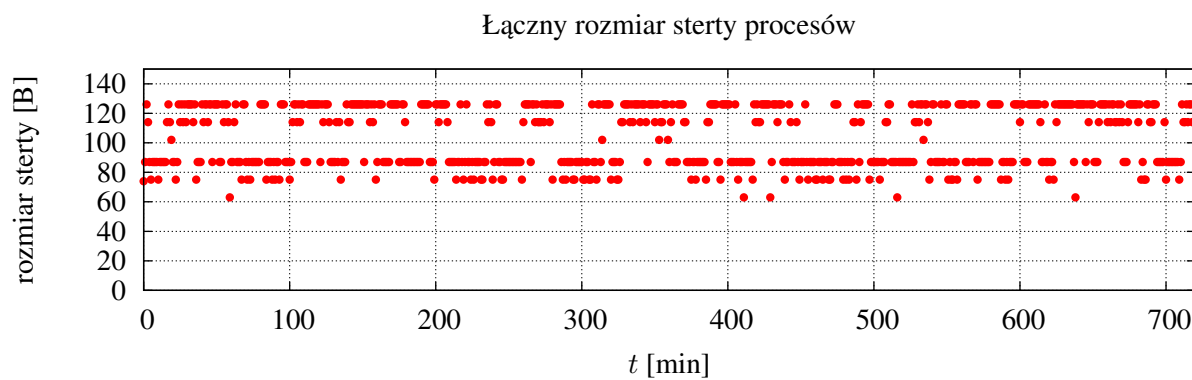
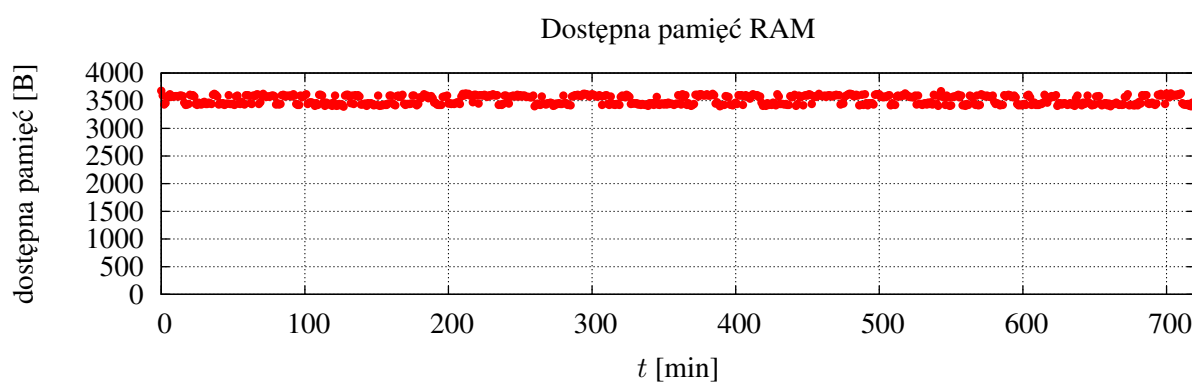
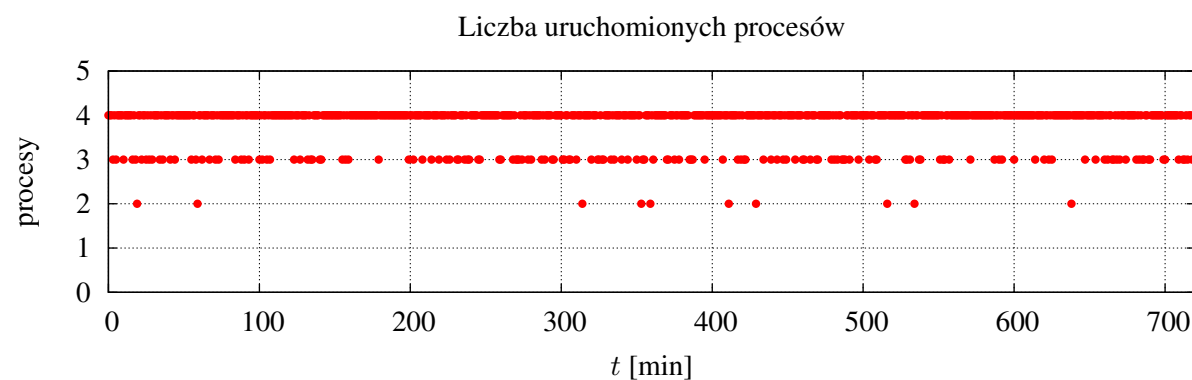
Szczególną uwagę należy zwrócić na rozmiar pamięci zajmowanej przez takie elementy maszyny wirtualnej jak tablica atomów czy załadowany kod modułów. Tablica atomów, łącznie z ok. 50 atomami załadowanymi do pamięci po starcie maszyny wirtualnej które potrzebne są do umieszczenia funkcji wbudowanych w tablicy eksportowanych funkcji, zajmuje prawie 4 kB pamięci RAM. Kod modułów, a w szczególności modułu `led_sup` zajął w pamięci prawie 6 kB pomimo niezbyt skomplikowanej logiki. Wprowadza on jednak do systemu pewne złożone wyrażenia (jak lista diod) oraz znaczącą liczbę nowych atomów.

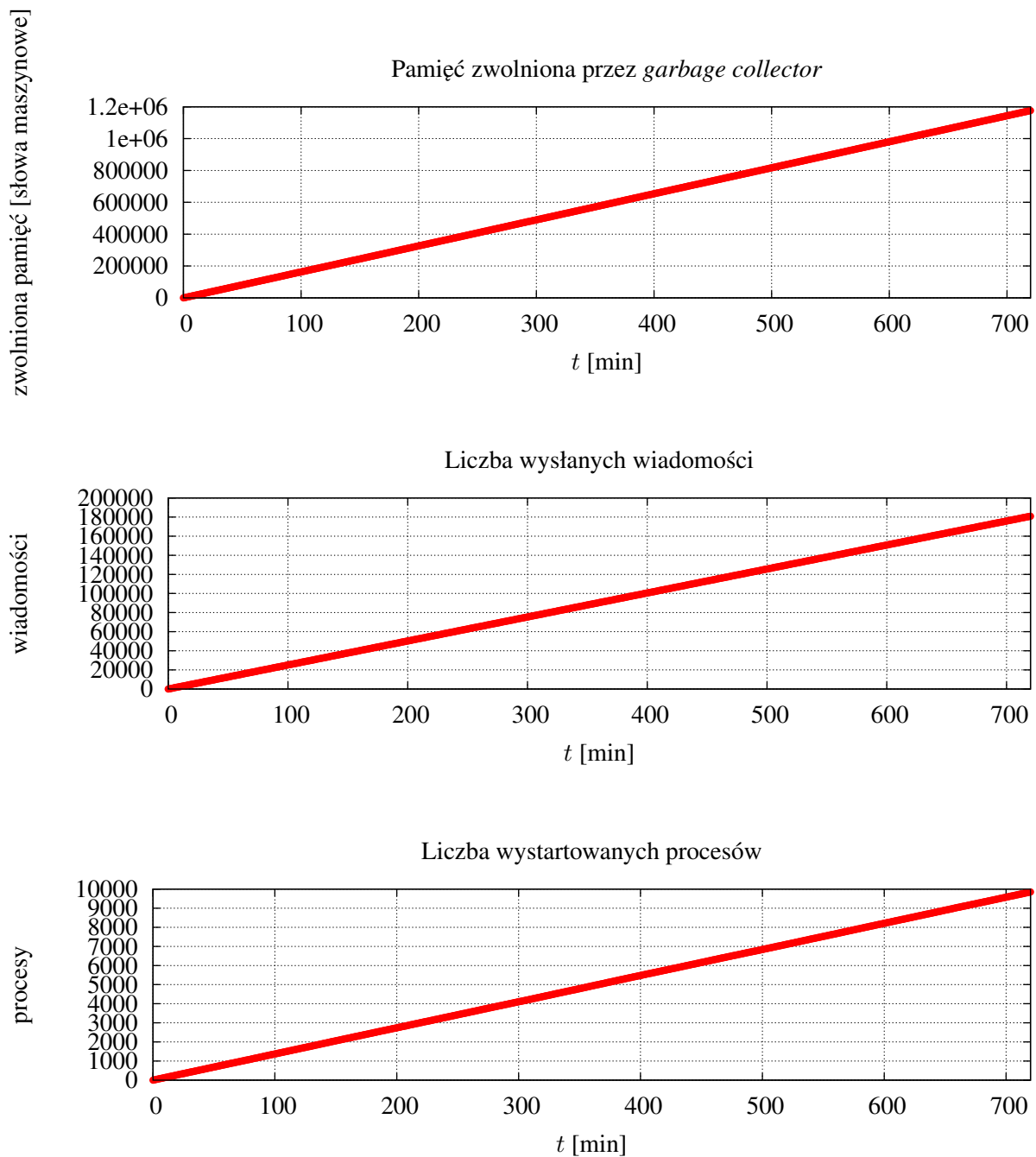
Do wystartowania całej maszyny wirtualnej z przykładową aplikacją potrzebne było ponad 22 kB pamięci z dostępnych 28 kB. Należy więc wysunąć wniosek, że tak ograniczony rozmiar pamięci RAM jak w przypadku mikrokontrolera LPC1769 będzie przeszkodą w implementacji kolejnych funkcji wbudowanych oraz programów o bardziej skomplikowanej logice, a co za tym idzie dłuższym kodzie pośrednim i z większą ilością uruchomionych procesów, aniżeli w przypadku opisanych w niniejszym rozdziale przykładowych aplikacji.

## 1.3. Sterownik RFM73

### 1.3.1. Cel aplikacji

Celem aplikacji było zaimplementowanie biblioteki sterownika do modułu radiowego RFM73 [10], produkowanego przez firmę Hope Microelectronics. Ten tani układ (koszt jednej sztuki to ok. 10 PLN) pozwala na komunikację bezprzewodową w paśmie 2,4 GHz z szybkością dochodzącą do 2 Mbps. Warstwa sprzętowa zapewnia możliwość wysłania pakietu składającego się z maksymalnie 32 bajtów. Odpowiedzialność za implementację protokołu pozwalającego na wysyłanie dłuższych pakietów należy już jednak do programisty.





Rysunek 1.4: Wyniki uruchomienia aplikacji kontrolującej stan diod LED za pomocą dedykowanych procesów.



Moduł komunikuje się ze światem zewnętrznym dzięki wbudowanemu w niego mikrokontrolerowi, za pomocą interfejsu szeregowego SPI (*Serial Peripheral Interface*). Sam układ zapewnia także sprawdzanie poprawności pakietu za pomocą sumy kontrolnej, prosty mechanizm retransmisji czy adresowania urządzeń w sieci.

Typowymi zastosowaniami modułu mogą być takie urządzenia jak np:

- urządzenia zdalnego sterowania;
- sensory przesyłające dane pomiarowe;
- urządzenia peryferyjne do komputerów osobistych.

Implementowana aplikacja miała pozwolić na przykładowe obsłużenie urządzenia peryferyjnego w języku Erlang, w tym wypadku za pomocą zaimplementowanych funkcji wbudowanych obsługujących interfejs SPI. Udostępnione funkcje wyprowadzeń z modułu RFM73 pozwoliły również na zaimplementowanie biblioteki w ten sposób, by możliwe było przetestowanie tłumaczenia przerwań zewnętrznych zgłaszanych do mikrokontrolera na wiadomości wysyłane do procesów.

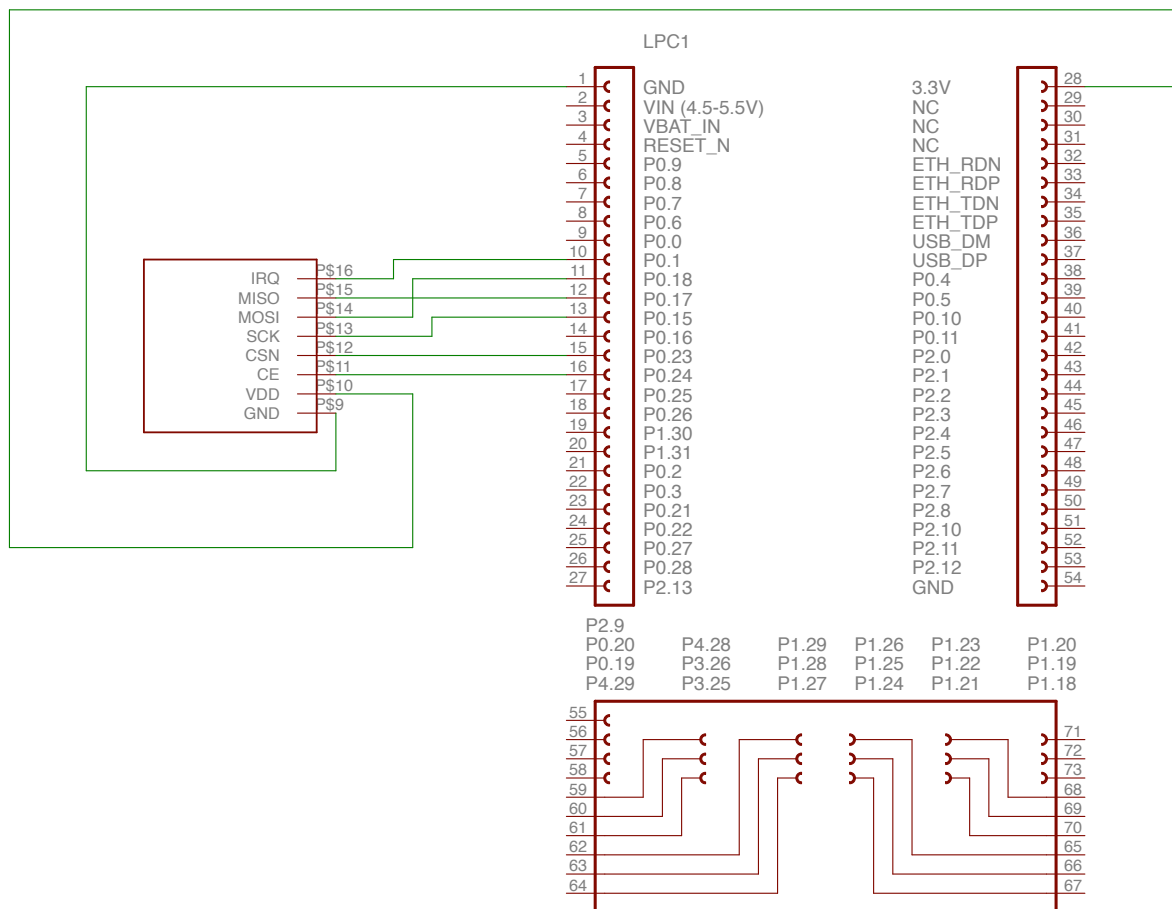
Fizyczna realizacja połączeń układu do płytki uruchomieniowej z mikrokontrolerem LPC1769, na którym uruchamiany był niniejszy przykład została zaprezentowana na rysunku 1.5.

### 1.3.2. Uzyskane wyniki

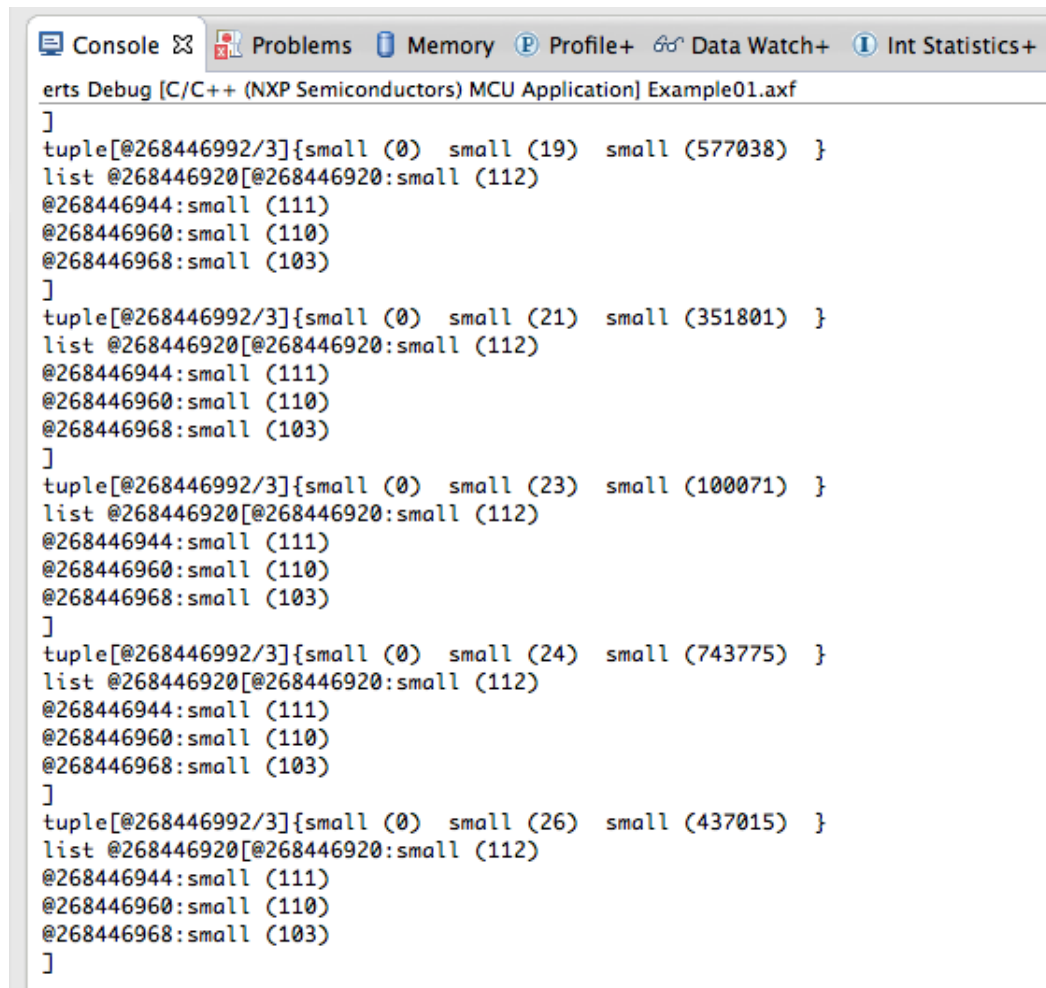
Implementowany sterownik miał zapewnić abstrakcję modułu RFM73 z poziomu aplikacji napisanej w języku Erlang z interfejsem pozwalającym na łatwe wysyłanie i odbieranie wiadomości drogą radiową. W celu przetestowania działania modułu, wysłano zestaw krótkich pakietów do innego urządzenia z podłączonym modułem tego samego typu. Program odbierający wiadomości po drugiej stronie odpowiadał wiadomościami o tej samej treści, dzięki czemu zweryfikowano poprawność danych przesyłanych w dwie strony.

### 1.3.3. Wnioski

Zaimplementowana biblioteka może być również podstawą do implementacji protokołu *Distributed Erlang*, dzięki któremu możliwy byłby do uruchomienia klastr urządzeń komunikujących się za pomocą modułu RFM73.



Rysunek 1.5: Schemat podłączenia modułu RFM73 do płytki prototypowej z mikrokontrolerem LPC1769



The screenshot shows a debugger's console window with the title bar 'erts Debug [C/C++ (NXP Semiconductors) MCU Application] Example01.axf'. The console displays a series of network packet captures. Each capture is represented by a line starting with 'tuple[@268446992/3]{', followed by three 'small' values in parentheses, and a closing brace. Below each tuple line is a 'list' line starting with '@268446920[@268446920:small (112)', followed by three memory addresses and their corresponding 'small' values in parentheses. The packets are numbered 1 through 5, with the tuple values increasing by 2 in the second position and 2048 in the third position for each subsequent packet.

```
erts Debug [C/C++ (NXP Semiconductors) MCU Application] Example01.axf
]
tuple[@268446992/3]{small (0)  small (19)  small (577038)  }
list @268446920[@268446920:small (112)
@268446944:small (111)
@268446960:small (110)
@268446968:small (103)
]
tuple[@268446992/3]{small (0)  small (21)  small (351801)  }
list @268446920[@268446920:small (112)
@268446944:small (111)
@268446960:small (110)
@268446968:small (103)
]
tuple[@268446992/3]{small (0)  small (23)  small (100071)  }
list @268446920[@268446920:small (112)
@268446944:small (111)
@268446960:small (110)
@268446968:small (103)
]
tuple[@268446992/3]{small (0)  small (24)  small (743775)  }
list @268446920[@268446920:small (112)
@268446944:small (111)
@268446960:small (110)
@268446968:small (103)
]
tuple[@268446992/3]{small (0)  small (26)  small (437015)  }
list @268446920[@268446920:small (112)
@268446944:small (111)
@268446960:small (110)
@268446968:small (103)
]
```

Rysunek 1.6: Informacja w konsoli o wiadomościach postaci "ping" przez proces



## Bibliografia

- [1] Erlang Public License. <http://www.erlang.org/EPLICENSE>. [data dostępu: 21.08.2014].
- [2] Ericsson AB. Erlang Embedded Systems User's Guide, 1997.
- [3] Ericsson AB. Distributed Erlang. [http://www.erlang.org/doc/reference\\_manual/distributed.html](http://www.erlang.org/doc/reference_manual/distributed.html), 2014. [data dostępu: 21.03.2014].
- [4] Ericsson AB. Erlang External Term Format. [http://erlang.org/doc/apps/erts/erl\\_ext\\_dist.html](http://erlang.org/doc/apps/erts/erl_ext_dist.html), 2014. [data dostępu: 21.03.2014].
- [5] Ericsson AB. Errors and Error Handling. [http://erlang.org/doc/reference\\_manual/errors.html](http://erlang.org/doc/reference_manual/errors.html), 2014. [data dostępu: 17.08.2014].
- [6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [7] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, 2003.
- [8] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd., 2011.
- [9] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.
- [10] HopeRF Electronic. *RFM73 Datasheet*. Hope Microelectronics Co. Ltd., 2006.
- [11] Anton Ertl. Threaded Code. <http://www.complang.tuwien.ac.at/forth/threaded-code.html>, 2004. [data dostępu: 12.07.2014].
- [12] Jim Gray. *Why Do Computers Stop And What Can Be Done About It?*, 1985.
- [13] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
- [14] Maxim Kharchenko. Erlang on Xen, A quest to lower startup latency. *Erlang Factory SF Bay Area 2012, San Francisco*, 2012.

- 
- [15] Erlang Solutions Ltd. Erlang Embedded. <http://www.erlang-embedded.com>, 2013. [data dostępu: 17.03.2014].
- [16] James Mistry. FreeRTOS and Multicore. Master's thesis, University of York, United Kingdom, 2011.
- [17] J. Morrison. EA IFF 85: Standard for interchange format files. *Amiga ROM Kernel Reference Manual: Devices (3rd edition)*, Addison-Wesley, 1(99):1, 1985.
- [18] Plataformatec. Elixir Language. <http://elixir-lang.org/>, 2014. [data dostępu: 7.07.2014].
- [19] NXP Semiconductors. *LPC1769/68/67/66/65/64/63. Product data sheet*. NXP Semiconductors, N.V., 2014.
- [20] NXP Semiconductors. LPCXpresso IDE. <http://www.lpcware.com/lpcxpresso/home>, 2014. [data dostępu: 3.07.2014].
- [21] Peer Stritzinger. Full Metal Erlang. *Erlang User Conference 2013, Stockholm*, 2013.