

**Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie**

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

KATEDRA INFORMATYKI STOSOWANEJ



PRACA MAGISTERSKA

RAFAŁ STUDNICKI

**PODSTAWOWA FUNKCJONALNOŚĆ ERLANGA DLA SYSTEMU
FREERTOS**

PROMOTOR:
dr inż. Piotr Matyasik

Kraków 2014

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

.....

PODPIS

AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Engineering in
Biomedicine

DEPARTMENT OF APPLIED COMPUTER SCIENCE



MASTER OF SCIENCE THESIS

RAFAŁ STUDNICKI

**IMPLEMENTATION OF BASIC FEATURES OF ERLANG FOR
FREERTOS**

SUPERVISOR:
Piotr Matyasik Ph.D

Krakow 2014

Spis treści

1. Wprowadzenie	6
1.1. Programowanie i zastosowanie systemów wbudowanych	6
1.2. Wykorzystanie Erlanga w programowaniu systemów wbudowanych	8
1.3. Cele pracy	10
1.4. Zawartość pracy	10
2. System operacyjny FreeRTOS	11
2.1. Wprowadzenie	11
2.2. Zadania i planista (<i>scheduler</i>)	11
2.3. Kolejki	11
2.4. Przerwania	11
2.5. Zarządzanie zasobami	11
2.6. Zarządzanie pamięcią	11
2.7. FreeRTOS i LPC17xx	11
2.8. Podsumowanie	11
3. Język programowania Erlang	12
3.1. Wprowadzenie	12
3.2. System typów danych	12
3.3. Kompilacja kodu źródłowego	12
3.3.1. Wprowadzenie	12
3.3.2. Kod źródłowy	12
3.3.3. Preprocessing	13
3.3.4. Transformacje drzewa syntaktycznego	14
3.3.5. Kod pośredni (<i>bytecode</i>)	15
3.3.6. Plik binarny BEAM	17
3.3.7. Podsumowanie	18
A. Lista operacji maszyny wirtualnej BEAM	19
Bibliografia	20

1. Wprowadzenie

W rozdziale uwzględniono wstępne informacje dotyczące programowania urządzeń wbudowanych, a także opisano dotychczasowe wykorzystanie języków funkcyjnych w programowaniu takich urządzeń. Opisano w nim także cele oraz zawartość niniejszej pracy.

1.1. Programowanie i zastosowanie systemów wbudowanych

System wbudowany jest to system komputerowy będący zazwyczaj integralną częścią urządzenia zawierającego elementy sprzętowe i mechaniczne. W przeciwieństwie do komputerów ogólne przeznaczenia, których celem jest realizacja różnego rodzaju zadań w zależności od potrzeb ich użytkowników, systemy wbudowane realizują tylko jedno, konkretne zadanie.

W obecnych czasach, gdy dąży się do tego, by coraz większa liczba urządzeń powszechnego użytku była "inteligentna" i mogła spełniać swoją zadania całkowicie niezależnie od człowieka, systemy wbudowane wykorzystywane w coraz większej mierze. Przykładami zastosowań systemów wbudowanych mogą być np.:

- telefony komórkowe;
- centrale telefoniczne;
- sterowniki do robotów mechanicznych;
- sprzęt sterujący samolotami i rakietami;
- układy sterujące pracą silnika samochodowego, komputery pokładowe;
- systemy alarmowe, antywłamaniowe, przeciwpożarowe;
- sprzęt medyczny;
- sprzęt pomiarowy.

Systemy wbudowane najczęściej implementowane są w oparciu o mikrokontrolery, czyli scalone systemy mikroprocesorowe zawierające na jednym, zintegrowanym układzie scalonym oprócz mikroprocesora również pamięć RAM, programu, układy wejścia-wyjścia, układy licznikowe oraz kontrolery przerwań. Zintegrowanie wszystkich tych elementów na jednej płytce pozwala na redukcję rozmiaru i poboru mocy takiego układu.

Spośród architektur projektowania oprogramowania przeznaczonego do programowania urządzeń wbudowanych można wymienić:

1. kontrola programu w pętli
program kontrolowany jest w pojedynczej pętli, wewnątrz której podejmowane są decyzje o sterowaniu elementami sprzętowymi lub programowymi;
2. kontrola programu przez przerwania
konkretne zadania programu wywoływane są przez wewnętrzne (np. zegary) lub zewnętrzne (np. odbiór danych z portu szeregowego) przerwania. Architektura ta często mieszana jest z wykonywaniem programu w pętli. W takim podejściu zadania o wysokim priorytecie wywoływane są przez przerwania, natomiast zadania o niskim priorytecie wykonywane są w pętli;
3. wielozadaniowość z wywłaszczaniem
w tego typu systemach pomiędzy kodem programu a mikrokontrolerem znajduje się niskopoziomowe oprogramowanie (jądro) odpowiadające za przydzielanie czasu procesora dla wielu współbieżnych zadań, które mogą mieć różne priorytety wykonania. Planista (*scheduler*) decyduje także w którym momencie powinno zostać obsłużone przerwanie;
4. wielozadaniowość bez wywłaszczania
tego rodzaju architektura jest bardzo podobna do wielozadaniowości z wywłaszczaniem, jednak jądro nie dokonuje samodzielnych decyzji o przerwaniu wykonywania któregoś ze współbieżnych zadań lecz pozostawia tę decyzję programiście;
5. mikrojądro
jest rozszerzeniem systemów obsługujących wielozadaniowość bez wywłaszczania lub z wywłaszczaniem poprzez dodanie np. zarządzania pamięcią, mechanizmów synchronizacji czy komunikacji pomiędzy współbieżnymi zadaniami do funkcjonalności jądra. Przykładami mikrojąder mogą być np. FreeRTOS, Enea OSE czy RTEMS;
6. jądro monolityczne do funkcjonalności jądra dodaje funkcjonalności zapewniające komplet komunikacji z peryferiami systemu dodające do funkcjonalności np. system plików, stos TCP/IP do komunikacji sieciowej, czy sterowniki obsługi urządzeń zewnętrznych. Spośród systemów z jądrami monolitycznymi można wymienić takie systemy jak Embedded Linux czy Windows CE.

Można zauważyć, że wymienione architektury zostały uporządkowane względem złożoności projektowanego systemu, ale także pod względem złożoności występującego elementu pośredniego pomiędzy programowanym fizycznym urządzeniem a oprogramowaniem. Wraz ze wzrostem złożoności systemu rosną również wymagania sprzętowe konieczne do uruchomienia danego systemu, maleje jednak bezpośredni poziom kontroli programisty nad realizacją wymagań czasu rzeczywistego. W niniejszej pracy rozważane będą sposoby implementacji oprogramowania na systemy wbudowane w oparciu o mikrojądro. Przykładem takiego systemu jest FreeRTOS.

1.2. Wykorzystanie Erlanga w programowaniu systemów wbudowanych

Jim Gray w pracy *Why Do Computers Stop And What Can Be Done About It?* [2] na podstawie obserwacji procesu projektowania i budowy sprzętu wchodzącego w skład systemów komputerowych sformułował pewne postulaty dotyczące implementacji oprogramowania odpornego na błędy. Były one następujące:

1. oprogramowanie powinno być modularne, co powinno zostać zapewnione przez wyabstrahowanie logiki w procesach. Komunikacja między procesami powinna zostać zapewniona przez mechanizm przesyłania wiadomości;
2. propagacja błędów powinna być powstrzymywana tak szybko jak to tylko możliwe (*fail-fast*);
3. logika wykonywana przez procesy powinna być zduplikowana w całym systemie tak, aby możliwe było jej wykonanie pomimo błędu sprzętowego lub tymczasowego błędu innego modułu;
4. powinien zostać zapewniony mechanizm transakcyjny pozwalający na zachowanie spójności danych;
5. powinien zostać zapewniony mechanizm transakcyjny, który w połączeniu z duplikacją procesów ułatwi obsługę wyjątków i tolerowanie błędów oprogramowania.

Obserwacje te były motywacją dla czwórki inżynierów z firmy Ericsson AB - Bjarne Dackera, Joe Armstrong, Mike'a Williams i Roberta Virdinga do stworzenia nowej platformy umożliwiającej rozwój oprogramowania, spełniającego powyższe wymagania. Jak się później okazało, do zaspokojenia wszystkich wymienionych potrzeb konieczne było stworzenie nowego, dedykowanego języka programowania - Erlang, wraz z zestawem bibliotek - OTP (Open Telecom Platform).

Rozwiązanie to zapewniało realizację powyższych postulatów poprzez następujące cechy:

1. składnia języka pozwalająca na tworzenie krótszych i bardziej zwężonych programów w porównaniu do języków imperatywnych;
2. zarządzanie pamięcią przy pomocy *garbage collector*, co pozwala na zwolnienie programisty z ręcznego zarządzania pamięcią i wynikających z tego częstych błędów;
3. izolowane, lekkie i możliwe do szybkiego uruchomienia procesy, które nie mogą bezpośrednio oddziaływać na inne uruchomione w systemie;
4. współbieżne uruchomienie procesów;
5. możliwość wykrywania błędów w jednym procesie przez drugi (monitorowanie procesów);
6. możliwość zidentyfikowania błędu i podjęcia odpowiedniej akcji w jego efekcie;
7. możliwość podmiany kodu uruchamianego programu w locie;

8. niezawodna baza danych (*Mnesia*, wchodząca w skład OTP).

Należy zaznaczyć, że celem przyświecającym twórcom języka od samego początku było zastosowanie go w urządzeniach w budowanych, jak np. w centrali telekomunikacyjnej Ericsson AXD301, która po dzień dzisiejszy pozostaje tego typu urządzeniem o największej liczbie sprzedanych egzemplarzy.

Nie można jednak zapomnieć o tym, że język ten został zaprojektowany dla systemów o miękkich wymaganiach czasu rzeczywistego. Systemy tego typu charakteryzują się tym, że oczekiwane czasy odpowiedzi w tym systemie są rzędu milisekund a odstępstwa od oczekiwanego czasu odpowiedzi powodują tylko spadek jakości usług danego systemu. W przeciwieństwie do tego, systemy o twardych wymaganiach czasu rzeczywistego uznaje się w takich sytuacjach za niefunkcjonalne.

Aktualnie, najpopularniejszym i właściwie jedynym typem środowiska, używanym do uruchamiania maszyny wirtualnej Erlanga dla celów produkcyjnych są pełnoprawne systemy operacyjne (głównie GNU/Linux lub Unix) uruchamiane na fizycznych maszynach bądź w środowiskach zwirtualizowanych. Wiąże się z tym dość wysokie wymagania, zarówno pod względem zasobów sprzętowych (jak np. ilość dostępnej pamięci RAM) jak i funkcjonalności samego systemu operacyjnego (jak np. mechanizmy komunikacji międzyprocesowej), konieczne do uruchomienia w pełni funkcjonalnej dystrybucji maszyny wirtualnej.

Dystrybucja maszyny wirtualnej Erlanga BEAM (Bjorn/Bogdan Erlang Abstract Machine), która utrzymywana jest przez firmę Ericsson AB, umożliwia jednak uruchomienie jej w trybie wbudowanym na takich systemach operacyjnych jak VxWorks czy Embedded Solaris. Pierwszy z nich jest systemem operacyjnym czasu rzeczywistego, jednak maszyna wirtualna została przeniesiona na ten system tylko w zakresie pozwalającym na uruchomienie na niej centrali telekomunikacyjnej, a jej uruchomienie wymaga 32 MB pamięci RAM i 22 MB przestrzeni dyskowej. Z kolei uruchomienie Erlang/OTP na systemie Embedded Solaris wymaga 17 MB pamięci RAM i 80 MB przestrzeni dyskowej. Szczegóły dotyczące wersji maszyny wirtualnej na te systemy operacyjne mogą zostać znalezione w dokumentacji Erlang/OTP [1].

Oprócz tego, aktualnie rozwijany, otwartym projektem związanym z uruchomieniem Erlanga na systemach wbudowanych jest Embedded Erlang. Powstający on przy współpracy firmy Erlang Solutions Ltd. Skupia się on jednak na uruchomieniu maszyny wirtualnej na Raspberry Pi oraz Paralleli, które wymagają pełnej dystrybucji systemu operacyjnego Linux. Szczegóły dotyczące projektu można znaleźć pod adresem [4].

Można zatem zauważyć, że wymagania, jakich do działania potrzebują zarówno wymienione przeniesienia (*porty*) maszyny BEAM oraz Embedded Erlang są zdecydowanie zbyt wysokie w porównaniu do specyfikacji sprzętowych rozważanych w niniejszej pracy.

W momencie powstawania pracy firma Ericsson AB była w trakcie implementacji maszyny wirtualnej Erlanga dla systemu operacyjnego czasu rzeczywistego Enea OSE. System ten abstrahuje logikę implementowanego oprogramowania w izolowanych procesach, komunikujących się między sobą poprzez wiadomości (*actor model*). Poziom zgodności funkcjonalności udostępnianych przez system z architekturą maszyny wirtualnej Erlanga sprawia, że OSE wydaje się być idealnym systemem do implementacji maszyny wirtualnej dla tego języka. Pozostaje on jednak produktem zamkniętym.

Innym projektem godnym uwagi jest Grisp, autorstwa Peera Stritzingera, będący portem maszyny wirtualnej Erlanga dla mikrojądra RTEMS [6]. W momencie pisania pracy Grisp również pozostaje w trakcie rozwoju, jednak tak jak i maszyna dla systemu OSE pozostaje projektem zamkniętym.

Warto także wspomnieć o projekcie Erlang on Xen autorstwa Maxima Kharchenki [3], którego celem jest zbudowanie wersji maszyny wirtualnej, której możliwe byłoby uruchomienie w środowisku zwirtualizowanym, bezpośrednio przez *hypervisor Xen*, bez systemu operacyjnego jako warstwy pośredniej. Aby cel mógł zostać osiągnięty, konieczna jest reimplementacja funkcjonalności maszyny wirtualnej przy jednoczesnym dopasowaniu ich do architektury hypervisora.

1.3. Cele pracy

Oczekiwanym efektem niniejszej pracy jest implementacja funkcjonalności systemu uruchomieniowego dla funkcyjnego, współbieżnego języka programowania Erlang dla systemu operacyjnego czasu rzeczywistego (mikrojądra) FreeRTOS. Zakres implementacji powinien pozwolić na uruchomienie kodu pośredniego (bajtkodu) maszyny wirtualnej Erlanga skompilowanego przez kompilator maszyny wirtualnej BEAM na mikrokontrolerach o ograniczonych zasobach sprzętowych (jak np. mikrokontroler z serii LPC17xx, mający 512kB pamięci flash i 64kB pamięci RAM).

Sposób implementacji powinien pozwolić na uruchamianie programów w taki sposób, by możliwe było spełnienie przynajmniej pierwszych czterech cech charakterystycznych dla języka Erlang z podrozdziału 1.2. Punkt 6. został na tym etapie pominięty - integracja systemu FreeRTOS z obsługą systemu plików leży poza zakresem pracy. Udostępnienie interfejsów sieciowych oraz możliwość korzystania z mechanizmu klastra Erlanga (*Distributed Erlang*) również nie jest jednym z celów niniejszej pracy.

Celem pracy jest są zatem:

- umożliwienie implementacji oprogramowania uruchamianego w ramach systemach wbudowanych o miękkich wymaganiach czasu rzeczywistego, przy pomocy funkcyjnego języka programowania Erlang;
- zbadanie wydajności rozwiązania na podstawie przykładowych programów uruchomionych na zaimplementowanej maszynie;
- udokumentowanie sposobu implementacji poszczególnych funkcjonalności i wskazanie różnic z implementacją oryginalnej maszyny wirtualnej;
- zwrócenie uwagi na możliwe kolejne kroki w implementacji maszyny wirtualnej opisanej w niniejszej pracy.

1.4. Zawartość pracy

2. System operacyjny FreeRTOS

W niniejszym rozdziale opisano architekturę systemu (mikrojądra) FreeRTOS wraz ze sposobem, w jaki poszczególne funkcjonalności mogą być przydatne w implementacji maszyny wirtualnej Erlanga dedykowanej dla tego systemu.

2.1. Wprowadzenie

Podstawową częścią każdego systemu operacyjnego jest jego jądro FreeRTOS jest mikrojądrem, umożliwiającym tworzenie aplikacji czasu rzeczywistego (zarówno o miękkich jak i twardych wymaganiach) przeznaczonych na systemy wbudowane.

2.2. Zadania i planista (*scheduler*)

2.3. Kolejki

2.4. Przerwania

2.5. Zarządzanie zasobami

2.6. Zarządzanie pamięcią

2.7. FreeRTOS i LPC17xx

Mikrokontroler LPC17xx jest jednym z systemów, na który przeniesiony został mikrojądrem FreeRTOS.

2.8. Podsumowanie

3. Język programowania Erlang

3.1. Wprowadzenie

3.2. System typów danych

Tu będzie coś o typach danych w Erlangu i ich reprezentacji w pamięci.

3.3. Kompilacja kodu źródłowego

Podrozdział opisuje kolejne kroki, z jakich składa się proces otrzymywania skompilowanego kodu pośredniego maszyny wirtualnej BEAM z kodu źródłowego napisanego w języku Erlang.

3.3.1. Wprowadzenie

Jak zostało wspomniane w ??, program napisany w języku Erlang wykonywany jest na dedykowanej do tego celu maszynie wirtualnej.

Narzędzia przeznaczone do operacji opisanych w niniejszym rozdziale zostały napisane w języku Erlang i dostępne są w aplikacji **compiler** dostarczanej wraz z maszyną wirtualną BEAM.

3.3.2. Kod źródłowy

```
1 | -module(fac) .
2 |
3 | -export([fac/1]) .
4 | -define(ERROR, "Invalid argument") .
5 |
6 | -include("fac.hrl") .
7 |
8 | fac(#factorial{n=0, acc=Acc}) ->
9 |     Acc;
10 | fac(#factorial{n=N, acc=Acc}) ->
11 |     fac(#factorial{n=N-1, acc=N*Acc});
12 | fac(N) when is_integer(N) ->
13 |     fac(#factorial{n=N});
14 | fac(N) when is_binary(N) ->
15 |     fac(binary_to_integer(N));
```

```

16 | fac(_) ->
17 |     {error, ?ERROR}.

```

Listing 3.1: Plik fac.erl

```

1 | -record(factorial, {n, acc=1}).

```

Listing 3.2: Plik fac.hrl

3.3.3. Preprocessing

```

1 | -file("fac.erl", 1).
2 |
3 | -module(fac).
4 |
5 | -export([fac/1]).
6 |
7 | -file("fac.hrl", 1).
8 |
9 | -record(factorial, {n, acc = 1}).
10 |
11 | -file("fac.erl", 7).
12 |
13 | fac(#factorial{n = 0, acc = Acc}) ->
14 |     Acc;
15 | fac(#factorial{n = N, acc = Acc}) ->
16 |     fac(#factorial{n = N - 1, acc = N * Acc});
17 | fac(N) when is_integer(N) ->
18 |     fac(#factorial{n = N});
19 | fac(N) when is_binary(N) ->
20 |     fac(binary_to_integer(N));
21 | fac(_) ->
22 |     {error, "Invalid argument"}.

```

Listing 3.3: Moduł fac po pierwszym przetworzeniu

```

1 | -file("fac.erl", 1).
2 |
3 | -file("fac.hrl", 1).
4 |
5 | -file("fac.erl", 7).
6 |
7 | fac({factorial, 0, Acc}) ->
8 |     Acc;
9 | fac({factorial, N, Acc}) ->
10 |     fac({factorial, N - 1, N * Acc});
11 | fac(N) when is_integer(N) ->
12 |     fac({factorial, N, 1});

```

```

13 fac(N) when is_binary(N) ->
14     fac(binary_to_integer(N));
15 fac(_) ->
16     {error,"Invalid argument"}.
17
18 module_info() ->
19     erlang:get_module_info(fac).
20
21 module_info(X) ->
22     erlang:get_module_info(fac, X).

```

Listing 3.4: Moduł fac po drugim przetworzeniu

3.3.4. Transformacje drzewa syntaktycznego

```

1 [{attribute,1,file,{"fac.erl",1}},
2  {attribute,1,module,fac},
3  {attribute,5,export,[{fac,1}]},
4  {attribute,1,file,{"fac.hrl",1}},
5  {attribute,1,record,
6    {factorial,
7      [{record_field,1,{atom,1,n}},
8       {record_field,1,{atom,1,acc},{integer,1,1}}]}},
9  {attribute,9,file,{"fac.erl",9}},
10 {function,10,fac,1,
11   [{clause,10,
12     [{record,10,factorial,
13       [{record_field,10,{atom,10,n},{integer,10,0}},
14        {record_field,10,{atom,10,acc},{var,10,'Acc'}}]}],
15     [],
16     [{var,11,'Acc'}]},
17    {clause,12,
18      [{record,12,factorial,
19        [{record_field,12,{atom,12,n},{var,12,'N'}},
20         {record_field,12,{atom,12,acc},{var,12,'Acc'}}]}],
21      [],
22      [{call,13,
23        {atom,13,fac},
24        [{record,13,factorial,
25          [{record_field,13,
26            {atom,13,n},
27            {op,13,'-',{var,13,'N'}},{integer,13,1}},
28          {record_field,13,
29            {atom,13,acc},
30            {op,13,'*',{var,13,'N'}},{var,13,'Acc'}}]}]}],
31      {clause,14,
32        [{var,14,'N'}],
33        [{call,14,{atom,14,is_integer},{var,14,'N'}}]},
34      [{call,15,

```

```

35         {atom,15,fac},
36         [{record,15,factorial,
37           [{record_field,15,{atom,15,n},{var,15,'N'}}]}]}],
38     {clause,16,
39       [{var,16,'N'}],
40       [[{call,16,{atom,16,is_binary},{var,16,'N'}}]],
41       [{call,17,
42         {atom,17,fac},
43         [{call,17,{atom,17,binary_to_integer},{var,17,'N'}}]}]},
44     {clause,18,
45       [{var,18,'_'}],
46       [],
47       [{tuple,19,[{atom,19,error},{string,19,"Invalid argument"}]}]}],
48 {eof,20}]

```

Listing 3.5: Drzewo syntaktyczne modułu fac

3.3.5. Kod pośredni (bytecode)

```

1  {module, fac}. %% version = 0
2
3  {exports, [{fac,1},{module_info,0},{module_info,1}]}.
4
5  {attributes, []}.
6
7  {labels, 11}.
8
9
10 {function, fac, 1, 2}.
11   {label,1}.
12   {line,[{location,"fac.erl",8}]}.
13   {func_info,{atom,fac},{atom,fac},1}.
14   {label,2}.
15   {test,is_tuple,{f,4},{x,0}}.
16   {test,test_arity,{f,4},{x,0},3}.
17   {get_tuple_element,{x,0},0,{x,1}}.
18   {get_tuple_element,{x,0},1,{x,2}}.
19   {get_tuple_element,{x,0},2,{x,3}}.
20   {test,is_eq_exact,{f,4},{x,1},{atom,factorial}}.
21   {test,is_eq_exact,{f,3},{x,2},{integer,0}}.
22   {move,{x,3},{x,0}}.
23   return.
24   {label,3}.
25   {line,[{location,"fac.erl",11}]}.
26   {gc_bif,'-',{f,0},4,[{x,2},{integer,1}],{x,0}}.
27   {line,[{location,"fac.erl",11}]}.
28   {gc_bif,'*',{f,0},4,[{x,2},{x,3}],{x,1}}.
29   {test_heap,4,4}.
30   {put_tuple,3,{x,2}}.

```

```

31     {put,{atom,factorial}}.
32     {put,{x,0}}.
33     {put,{x,1}}.
34     {move,{x,2},{x,0}}.
35     {call_only,1,{f,2}}.
36 {label,4}.
37     {test,is_integer,{f,5},{x,0}}.
38     {test_heap,4,1}.
39     {put_tuple,3,{x,1}}.
40     {put,{atom,factorial}}.
41     {put,{x,0}}.
42     {put,{integer,1}}.
43     {move,{x,1},{x,0}}.
44     {call_only,1,{f,2}}.
45 {label,5}.
46     {test,is_binary,{f,6},{x,0}}.
47     {allocate,0,1}.
48     {line,[{location,"fac.erl",15}]}.
49     {call_ext,1,{extfunc,erlang,binary_to_integer,1}}.
50     {call_last,1,{f,2},0}.
51 {label,6}.
52     {move,{literal,{error,"Invalid argument"}},{x,0}}.
53     return.
54
55
56 {function, module_info, 0, 8}.
57     {label,7}.
58     {line,[]}.
59     {func_info,{atom,fac},{atom,module_info},0}.
60 {label,8}.
61     {move,{atom,fac},{x,0}}.
62     {line,[]}.
63     {call_ext_only,1,{extfunc,erlang,get_module_info,1}}.
64
65
66 {function, module_info, 1, 10}.
67     {label,9}.
68     {line,[]}.
69     {func_info,{atom,fac},{atom,module_info},1}.
70 {label,10}.
71     {move,{x,0},{x,1}}.
72     {move,{atom,fac},{x,0}}.
73     {line,[]}.
74     {call_ext_only,2,{extfunc,erlang,get_module_info,2}}.

```

Listing 3.6: Bytecode modułu fac

3.3.6. Plik binarny BEAM

Efekt przetworzenia kodu pośredniego, wyrażonego w postaci krotek, jest plik binarny w formacie IFF [5], w formacie zrozumiałym przez maszynę wirtualną BEAM. Maszyna ta wykorzystuje tego rodzaju pliki do ładowania kodu modułów do pamięci. Ich źródłem może być zarówno system plików na fizycznej maszynie, na której uruchomiony został BEAM, jak i inna maszyna wirtualna znajdująca się w tym samym klastrze *Distributed Erlang*, co docelowa.

W tabeli 3.1 zaprezentowana została struktura pliku binarnego maszyny wirtualnej BEAM.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	"FOR1"															
4	32	Rozmiar pliku bez pierwszych 8 bajtów															
8	64	"BEAM"															
12	96	Identyfikator fragmentu (<i>chunk</i>) 1															
16	128	Rozmiar fragmentu 1															
20	160	Dane fragmentu 1															
...	...	Identyfikator fragmentu (<i>chunk</i>) 2															
...															

Tablica 3.1: Struktura pliku kodu pośredniego BEAM

Każdy plik binarny BEAM powinien zawierać przynajmniej 5 następujących fragmentów *chunków* (obok opisu każdego fragmentu, w nawiasie podano tabelę, która reprezentuje strukturę dane fragmentu w pliku BEAM):

- tablica atomów wykorzystywanych przez moduł;
- *bytecode* danego modułu;
- tablica stringów wykorzystywanych przez moduł;
- tablica funkcji importowanych przez moduł;
- tablica funkcji eksportowanych przez moduł.

Ponadto, w pliku mogą znajdować się następujące, opcjonalne fragmenty:

- tablica lamdb wykorzystywanych przed moduł (tabela 3.2);

- tablica stałych wykorzystywanych przed modułem;
- lista atrybutów modułu;
- lista dodatkowych informacji o kompilacji modułu;
- tablica linii kodu źródłowego modułu (używana przy debuggowaniu i generacji stosu wywołań - *stacktrace*);
- drzewo syntaktyczne pliku z kodem źródłowym.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	"FOR1"															
4	32	Rozmiar pliku bez pierwszych 8 bajtów															
8	64	"BEAM"															
12	96	Identyfikator fragmentu (<i>chunk</i>) 1															
16	128	Rozmiar fragmentu 1															
20	160	Dane fragmentu 1															
...	...	Identyfikator fragmentu (<i>chunk</i>) 2															
...															

Tablica 3.2: Struktura tablicy atomów w pliku BEAM

W przypadku każdego rodzaju fragmentu, obszar jaki zajmuje on w pliku jest zawsze wielokrotnością 4 bajtów. Nawet jeżeli nagłówek fragmentu, zawierający jego rozmiar, nie jest podzielny przez 4, obszar zaraz za danym fragmentem dopełniany jest zerami do pełnych 4 bajtów.

Warto zaznaczyć również, że sposób implementacji maszyny wirtualnej BEAM nie definiuje kolejności w jakiej fragmenty powinny występować w pliku binarnym.

3.3.7. Podsumowanie

A. Lista operacji maszyny wirtualnej BEAM

Dodatek zawiera listę operacji maszyny wirtualnej BEAM, jakie może zawierać skompilowany kod pośredni przez nią wykonywany. Lista zawiera nazwę operacji, jej argumenty oraz opis jej działania.

Kod operacji oraz każdy argument zajmują zawsze 1 bajt w pliku skompilowanego kodu pośredniego. Kolejność bajtów w zapisie kodu pośredniego to *big endian*.

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji
szesnastkowo	dziesiętnie		
01	1	label Lbl	Wprowadza lokalną dla danego modułu etykietę identyfikującą aktualne miejsce w kodzie.
02	2	func_info M F A	Definiuje funkcję F, w module M o arności A.
03	3	int_code_end	???

Tablica A.1: Lista operacji maszyny wirtualnej BEAM

Bibliografia

- [1] Ericsson AB. Erlang embedded systems user's guide, 1997.
- [2] Jim Gray. Why do computers stop and what can be done about it?, 1985.
- [3] Maxim Kharchenko. Erlang on xen, a quest to lower startup latency. *Erlang Factory SF Bay Area 2012, San Francisco*, 2012.
- [4] Erlang Solutions Ltd. Erlang embedded.
- [5] J. Morrison. Ea iff 85: Standard for interchange format files. *Amiga ROM Kernel Reference Manual: Devices (3rd edition)*, Addison-Wesley, 1(99):1, 1985.
- [6] Peer Stritzinger. Full metal erlang. *Erlang User Conference 2013, Stockholm*, 2013.