



**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 poświadczanie 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. Wprowadzenie</b>	9
1.1. Programowanie i zastosowanie systemów wbudowanych	9
1.2. Wykorzystanie Erlanga w programowaniu systemów wbudowanych	11
1.3. Cele pracy	13
1.4. Zawartość pracy	14
<b>2. System operacyjny FreeRTOS</b>	15
2.1. Zadania i planista ( <i>scheduler</i> )	15
2.2. Kolejki	16
2.3. Przerwania	17
2.4. Zarządzanie zasobami	17
2.5. Zarządzanie pamięcią	18
2.6. FreeRTOS i LPC176x	18
2.7. Podsumowanie	20
<b>3. Język programowania Erlang</b>	21
<b>4. Zaimplementowane elementy maszyny wirtualnej</b>	23
4.1. Moduł ładujący kod ( <i>loader</i> )	23
4.2. Tablice	24
4.2.1. Tablica atomów	26
4.2.2. Tablica eksportowanych funkcji	26
4.3. Typy danych	27
4.3.1. Wartości bezpośrednie a pośrednie	27
4.3.2. Wartości bezpośrednie	29
4.3.3. Listy	29
4.3.4. Krotki	30
4.3.5. Duże liczby	31
4.3.6. Niezaimplementowane typy danych	32
4.4. Interpreter kodu maszynowego	33

4.4.1.	Maszyna stosowa a rejestrowa .....	33
4.4.2.	Model interpretera.....	34
4.4.3.	Sposób implementacji .....	35
4.5.	Procesy .....	37
4.5.1.	Tablica procesów .....	37
4.5.2.	Struktura procesu .....	37
4.5.3.	Komunikacja międzyprocesowa.....	38
4.5.4.	Obsługa błędów.....	39
4.6.	Planista ( <i>scheduler</i> ) .....	40
4.7.	Funkcje wbudowane ( <i>Built-In Functions</i> ) .....	42
4.7.1.	Funkcje ogólne (moduł <i>erlang</i> ).....	43
4.7.2.	Listy (moduł <i>lists</i> ).....	44
4.7.3.	GPIO i obsługa przerwań (moduł <i>lpc_gpio</i> ).....	44
4.7.4.	SPI (moduł <i>lpc_spi</i> ).....	44
4.7.5.	Funkcje pomocnicze (moduł <i>lpc_debug</i> ).....	44
4.8.	<i>Garbage collector</i> .....	44
4.8.1.	Pamięć zajmowana przez proces.....	45
4.8.2.	Algorytm Cheney'a .....	46
4.8.3.	Podejście generacyjne w maszynie BEAM.....	50
4.9.	Mechanizmy zarządzania czasem.....	51
4.10.	Podsumowanie różnic z maszyną BEAM.....	53
<b>5.</b>	<b>Przykładowe aplikacje</b> .....	<b>55</b>
5.1.	Silnia.....	55
5.1.1.	Cel aplikacji .....	55
5.1.2.	Uzyskane wyniki.....	56
5.1.3.	Wnioski .....	57
5.2.	Kontrola diod LED przez procesy .....	59
5.2.1.	Cel aplikacji .....	59
5.2.2.	Uzyskane wyniki.....	60
5.2.3.	Wnioski .....	60
5.3.	Sterownik RFM73 .....	60
5.3.1.	Cel aplikacji .....	60
5.3.2.	Uzyskane wyniki.....	61
5.3.3.	Wnioski .....	62

<b>6. Podsumowanie</b>	63
6.1. Wnioski	63
6.2. Dalszy rozwój projektu	63
<b>A. Zawartość dołączonej płyty CD</b>	65
<b>B. Kompilator aplikacji</b>	67
B.1. Opis aplikacji	67
B.2. Kompilacja narzędzia	67
B.3. Użycie narzędzia	68
<b>C. Konfiguracja parametrów maszyny wirtualnej</b>	69
<b>D. Kompilacja kodu źródłowego</b>	71
D.1. Wprowadzenie	71
D.2. Kod źródłowy	71
D.3. Preprocessing	72
D.4. Drzewo składniowe	73
D.5. Core Erlang	75
D.6. Kod pośredni - bajtkod	76
D.7. Plik binarny BEAM	79
D.7.1. Tablica atomów	80
D.7.2. Kod pośredni	81
D.7.3. Tablica importowanych funkcji	81
D.7.4. Tablica eksportowanych funkcji	82
D.7.5. Tablica funkcji lokalnych	82
D.7.6. Tablica lambda	83
D.7.7. Tablica stałych	84
D.7.8. Lista atrybutów modułu	85
D.7.9. Lista dodatkowych informacji o kompilacji modułu	85
D.7.10. Tablica linii kodu źródłowego modułu	85
D.7.11. Drzewo syntaktyczne modułu	86
<b>E. Lista instrukcji maszyny wirtualnej BEAM</b>	87
E.1. Typy argumentów	87
E.2. Lista instrukcji	89
<b>Bibliografia</b>	105





# 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 swoje zadania całkowicie niezależnie od człowieka, systemy wbudowane wykorzystywane są w coraz większej mierze. Przykładami zastosowań systemów wbudowanych mogą być np.:

- telefony komórkowe;
- centrale telefoniczne;
- sterowniki do robotów mechanicznych;
- sprzęt sterujący samolotami i raketami;
- 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 pamięci RAM i ROM, układy wejścia-wyjścia, układy licznikowe oraz kontrolery

przerwań. Zintegrowanie wszystkich tych elementów na jednej płycie pozwala na redukcję rozmiaru i poboru mocy takiego układu.

Spośród architektur oprogramowania uruchamianego na urządzeniach wbudowanych można wymienić:

1. **kontrolę programu w pętli** - program kontrolowany jest w pojedynczej pętli, wewnątrz której podejmowane są decyzje o sterowaniu elementami sprzętowymi lub programowymi;
2. **kontrolę programu przez przerwania** - konkretne części 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 o tym, 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?* [11] 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. Jedynym sposobem komunikacji pomiędzy procesami powinien być 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 Williamsa i Roberta Virdinga do stworzenia nowej platformy umożliwiającej rozwój oprogramowania, spełniającego powyższe wymagania. Jak się później okazało, do zaspokojenia wszystkich wymienionych wyżej potrzeb konieczne było stworzenie nowego, dedykowanego, funkcyjnego 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 zwięzłych 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 [6].

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 reakcji na zdarzenie 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 rozwijanym, otwartym projektem związanym z uruchomieniem Erlanga na systemach wbudowanych jest Embedded Erlang [14], który powstaje nakładem sił firmy Erlang Solutions Ltd. Skupia się on jednak na uruchomieniu maszyny wirtualnej na platformach sprzętowych typu Raspberry Pi czy Parallela, które wymagają pełnej dystrybucji systemu operacyjnego Linux.

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* [12]). 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 [20]. 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 [13], 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 ponowna implementacja części 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, gdyż integracja systemu FreeRTOS z obsługą systemu plików leży poza zakresem pracy. Udostępnienie interfejsów sieciowych oraz możliwość korzystania z mechanizmu klastrowania 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

Niniejsza praca została podzielona na sześć rozdziałów i cztery dodatki.

W rozdziale 2 opisano funkcjonalności udostępniane przez system operacyjny czasu rzeczywistego FreeRTOS. Rozdział 3 opisuje cechy charakterystyczne Erlanga, zarówno jako funkcyjnego języka programowania jak i jego maszyny wirtualnej. W rozdziale 4 opisano funkcjonalności maszyny wirtualnej Erlanga, które zaimplementowano w ramach pracy i porównano je do sposobu działania maszyny BEAM. W rozdziale 5 opisano trzy przykładowe aplikacje zaimplementowane w języku Erlang, które zostały uruchomione na zaimplementowanej maszynie wirtualnej, ze szczególnym uwzględnieniem wyników działania programów. Rozdział 6 zawiera podsumowanie pracy z wnioskami a także z obszarami, które warto rozwinąć w ramach dalszej pracy nad maszyną.

W dodatku A opisano zawartość płyty CD dołączonej do niniejszej pracy. Dodatek B opisuje sposób działania narzędzia służącego do kompilacji kodu źródłowego w Erlangu i odpowiedniej konfiguracji maszyny wirtualnej tak, aby zawierała skompilowany kod pośredni dla tych modułów. W dodatku D opisane zostały kroki pośrednie, jakie wykonuje kompilator języka Erlang aby przejść z kodu źródłowego napisanego w tym języku do kodu pośredniego modułu. Zaprezentowana została w nim także struktura wyjściowego pliku procesu kompilacji. Dodatek E zawiera listę instrukcji, jakie mogą znaleźć się w pliku z kodem pośrednim wraz ze sposobem zapisu argumentów operacji.

## 2. System operacyjny FreeRTOS

Podstawową częścią każdego systemu operacyjnego jest jego jądro, które odpowiedzialne jest za udostępnianie zasobów sprzętowych, takich jak procesor, pamięć czy urządzenia wejścia/wyjścia programom wykonywanym na tym systemie.

System FreeRTOS jest mikrojądrem (por. architektury oprogramowania na str. 10), przy użyciu którego możliwa jest implementacja aplikacji czasu rzeczywistego (zarówno o miękkich jak i twardych wymaganiach) na urządzeniach wbudowanych.

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. Zadania i planista (*scheduler*)

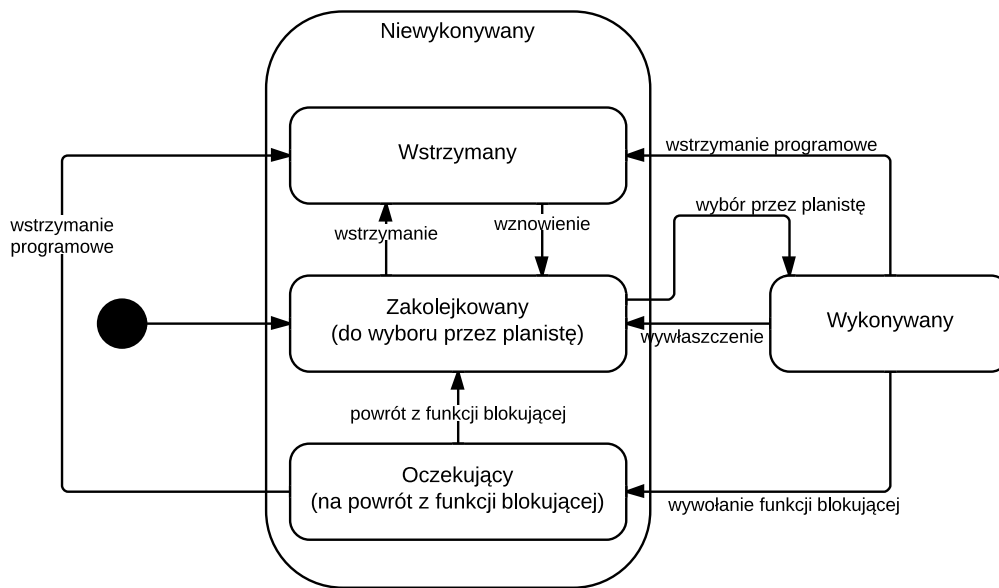
Podstawową wykonywalną jednostką w systemie FreeRTOS jest zadanie, zarządzane przez wbudowanego w system planistę (*scheduler*). Zadanie uruchomione pod nadzorem planisty można porównać do wątku w systemie Linux, z tą różnicą, że kod zadania musi zostać zaimplementowany w języku C i przed rozpoczęciem jego wykonywania należy zadeklarować rozmiar stosu danego zadania.

Zadaniom można również nadawać priorytety. Jeżeli do wykonania przeznaczone są zadania o różnych priorytetach, to w pierwszej kolejności wykonane zostanie to o wyższym priorytecie. W bardzo podobny sposób działa algorytm kolejowania procesów w maszynie wirtualnej BEAM.

Samo zadanie można znajdować się w kilku stanach w zależności m.in. od tego, czy planista wybrał je do wykonania, czy dopiero oczekuje ono na swoją kolej. Pełny diagram stanów, w jakich może znajdować się zadanie w systemie FreeRTOS zaprezentowany został na rysunku 2.1.

*Scheduler* może pracować w dwóch trybach: wyłączeniowym, w którym sam algorytm planisty decyduje o kolejności wykonywania zadań, oraz w trybie opartym na współpracy, w którym zadania „dobrowolnie” rezygnują z czasu procesora, który został im przydzielony. W tym drugim przypadku priorytety zadań są nadal brane pod uwagę podczas wyboru kolejnego zadania do wykonania.

Wielozadaniowość oparta na współpracy to model, jaki został zaimplementowany w oryginalnej maszynie wirtualnej Erlanga, po wprowadzeniu pojęcia redukcji jako miary czasu, przez jaki danemu procesowi udostępniona jest moc obliczeniowa (por. podrozdział 4.6).



Rysunek 2.1: Diagram stanów zadania w systemie FreeRTOS

Wymienione cechy charakterystyczne zadań i planisty stanowiły bardzo dobry punkt wyjścia do oparcia implementacji *schedulera* maszyny wirtualnej Erlanga na planiście systemu FreeRTOS oraz enkapsulację logiki procesów w zadaniach.

## 2.2. Kolejki

System FreeRTOS zapewnia mechanizm kolejki wiadomości między procesami, na wzór kolejki wiadomości POSIX. Kolejki nie należą do żadnego z zadań, dlatego też każde z zadań może zarówno odczytywać jak i zapisywać dane do każdej z nich. Proces przesłania i odebrania wiadomości polega na skopiowaniu danych z przestrzeni adresowej zadania-nadawcy do przestrzeni adresowej kolejki a następnie z przestrzeni adresowej kolejki do przestrzeni adresowej zadania-adresata.

Kolejki w systemie FreeRTOS bardzo dobrze oddają semantykę kolejki wiadomości (*mailbox*) w procesie erlangowym. Jednakże, aby oprzeć na nich implementację tej funkcjonalności, istniałaby konieczność utworzenia osobnej kolejki dla każdego z uruchomionych w systemie procesów, do czego konieczne jest z góry zaalokowanie pamięci dla kolejki wiadomości o maksymalnej długości.

W związku z tym, w maszynie wirtualnej opisanej w niniejszej pracy kolejki wiadomości zostały zaimplementowane wewnątrz zadań implementujących logikę procesów. Pozwoliło to na uproszczenie procedury wysłania wiadomości do procedury przez umieszczenie wiadomości na sterku procesu będącego jej adresatem, z której proces będzie mógł korzystać aż do momentu odśmiecenia pamięci procesu.



## 2.3. Przerwania

FreeRTOS zapewnia obsługę zarówno programowych jak i sprzętowych przerwań. Podejściem do implementacji obsługi przerwań, który zalecany jest przez autorów systemu jest ich odroczenie i delegacja obsługi do innego zadania, niż to obsługujące przerwanie (*Interrupt Service Routine* - ISR) [7]. Motywacją do tego, aby kod ISR był możliwie jak najkrótszy jest fakt, że w momencie jego wykonywania nowe przerwania nie są wykrywane.

W implementacji maszyny wirtualnej Erlanga dla FreeRTOS podążono za tą koncepcją i informacja o przerwaniu jest przesyłana jako wiadomość do procesów, które wywołały wcześniej odpowiednią wbudowaną funkcję subskrybującą. Efektem takiego wywołania jest zgłoszenie maszynie wirtualnej, że dany proces jest „zainteresowany” otrzymywaniem wiadomości dotyczących przerwań danego rodzaju.

## 2.4. Zarządzanie zasobami

W systemach, które pozwalają na działanie wielu zadań współbieżnie niezbędna jest obecność mechanizmów pozwalających na zarządzanie dostępem do pewnych obszarów pamięci. W sytuacji, gdy dwa współbieżne zadania (np. zadanie obsługujące przerwanie i zadanie implementujące logikę procesu) będą modyfikować pewien obszar pamięci w sposób nieatomiczny i jedno z zadań zostanie wyłączone w momencie, gdy cała operacja nie zostanie zakończona, obszar pamięci pozostanie w stanie niespójnym.

FreeRTOS zapewnia następujące mechanizmy do synchronizacji zadań:

- **sekcja krytyczna** - powoduje zablokowanie dostępu do czasu procesora dla wszystkich pozostałych zadań, możliwe jest także zablokowanie obsługi pewnego rodzaju przerwań;
- **mutex** - pozwalający na synchronizację dostępu do dzielonego zasobu przez „zainteresowane” zadania, które muszą uzyskać dostęp do mutexu. Ma do tego prawo tylko jedno zadanie w jednym czasie, przed wykonaniem operacji na dzielonym zasobie;
- **semafor** - działający jak mutex, pozwalający na dostęp większej liczby zadań do zasobu, jest zablokowany gdy jego wartość jest równa 0. Mutex jest szczególnym przypadkiem semafora, mogącym przyjąć tylko wartość 0 lub 1. W systemie FreeRTOS zarówno mutex jak i semafor zaimplementowane są przy użyciu tych samych struktur danych.

Wymienione mechanizmy synchronizacji zostały wykorzystane w elementach maszyny wirtualnej, w których było to konieczne, głównie w przypadku operacji wejścia-wyjścia. Należy wspomnieć, że ze względu na rozważany typ maszyny (uruchamianej na procesorze o jednym rdzeniu) jak i również ze względu na model wielozadaniowości oparty na współpracy, udało się uniknąć użycia mechanizmów synchronizacji w wielu miejscach, w których intuicja podpowiadałaby ich użycie.

## 2.5. Zarządzanie pamięcią

FreeRTOS udostępnia, spójny dla wszystkich swoich portów, interfejs do zarządzania pamięcią, składający się z dwóch funkcji: `pvPortMalloc()` oraz `vPortFree()` będące odpowiednikami funkcji systemowych `malloc()` i `free()`.

Programista implementujący aplikację z użyciem mikrojądra FreeRTOS może wybrać jedną z czterech implementacji powyższych funkcji:

- **heap\_1** - która nie pozwala na zwolnienie raz zaalokowanej pamięci, przeznaczona do bardzo prostych aplikacji wbudowanych, w których liczba i rozmiar struktura jest z góry znana;
- **heap\_2** - używająca algorytmu najlepszego dopasowania (*best-fit*) do alokacji bloku pamięci, nie pozwala jednak na ponowne użycie dwóch sąsiednich, zwolnionych wcześniej bloków do zaalokowania nowego, większego bloku;
- **heap\_3** - opakowująca wewnątrz sekcji krytycznej funkcje `malloc` i `free` udostępnianie przez kompilator, wadą tego rozwiązania jest duży rozmiar pliku wynikowego;
- **heap\_4** - działająca jak **heap\_2**, rozwiązująca jednak problem alokacji pamięci w sąsiednich blokach.

Mając na uwadze fakt, że implementowane środowisko uruchomieniowe przeznaczone będzie dla języka z automatycznym zarządzaniem pamięcią, wraz ze wzrostem rozmiaru pamięci procesu algorytm *garbage collector* będzie potrzebował alokować coraz to większe bloki pamięci. Z tego powodu należało zwrócić szczególną uwagę na wybór strategii zapewniającą dobrą fragmentację dostępnej pamięci RAM, wybraną strategią zarządzania pamięcią do użycia w implementowanej maszynie wirtualnej została zatem implementacja **heap\_4**.

Oryginalna maszyna wirtualna Erlanga używa wielu różnych strategii alokowania pamięci, minimalizujących jej fragmentację, w zależności od przeznaczenia danego segmentu pamięci i jego rozmiaru. Maszyna rozważana w niniejszej pracy używa tylko ww. interfejsu udostępnionego przez FreeRTOS i jemu powierza zadanie dobrego dopasowania alokowanych obszarów pamięci.

## 2.6. FreeRTOS i LPC176x

Mikrojądro FreeRTOS zostało przeniesione na ponad 20 rodzin mikrokontrolerów, w tym na LPC1769, który zawiera procesor ARM Cortex-M3.

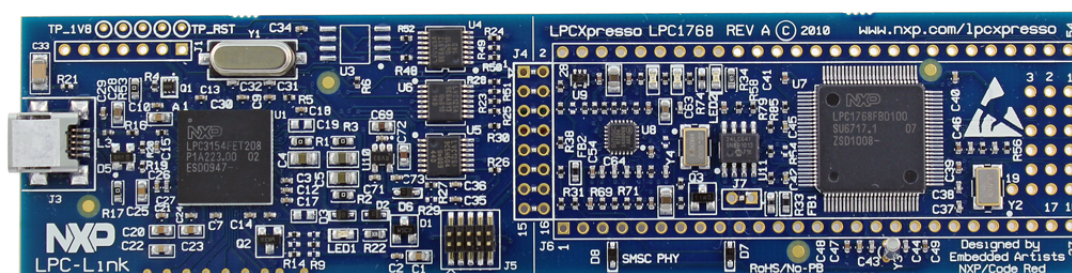
Mikrokontroler ten ma następujące parametry:

- 64 kB pamięci SRAM;
- 512 kB pamięci flash;
- posiada 4 interfejsy UART (*Universal Asynchronous Receiver-Transmitter*);

- posiada 3 interfejsy I<sup>2</sup>C / TWI (*Two-Wire Interface*);
- posiada 1 interfejs SPI (*Serial Peripheral Interface*);
- posiada 2 interfejsy SSP (*Synchronous Serial Port*);
- posiada 2 interfejsy CAN (*Controller Area Network*);
- posiada interfejs modulacji sygnału cyfrowego PWM (*Pulse-Width Modulation*);
- posiada 1 interfejs USB 2.0 (*Universal Serial Bus*);
- posiada 70 pinów ogólnego przeznaczenia GPIO (*General Purpose Input-Output*);
- posiada 12-bitowy przetwornik analogowo-cyfrowy (ADC);
- posiada 10-bitowy przetwornik cyfrowo-analogowy (DAC);
- posiada cztery liczniki ogólnego przeznaczenia;
- posiada zegar czasu rzeczywistego (RTC) z dedykowanym dla niego źródłem sygnału zegarowego.

Dokładne właściwości wymienionych wyżej elementów mikrokontrolera zawarte są w jego nocie katalogowej [18].

W sprzedaży dostępna jest tania płytką rozwojowa ze wspomnianym mikrokontrolerem, produkowana przez firmę NXP, posiadająca zintegrowany interfejs JTAG, służący do debugowania aplikacji. Wygląd takiego układu uruchomieniowego został zaprezentowany na rysunku 2.2. Przez tę samą firmę udostępniane jest również środowisko deweloperskie pozwalające na łatwą kompilację i debugowanie rozwijanego oprogramowania - LPCXpresso.



Rysunek 2.2: Płytkę rozwojowa z mikrokontrolerem LPC176x. Źródło: <http://www.embeddedartists.com/>

Maszyna wirtualna opisywana w niniejszej pracy została rozwijana na ww. mikrokontrolerze, z użyciem ww. narzędzi deweloperskich. Można założyć, że maszyna wirtualna Erlanga prezentowana w niniejszej pracy, po dopasowaniu działania niektórych funkcji wbudowanych do architektury odpowiedniego mikrokontrolera, będzie działać z portami systemu FreeRTOS również na inne platformach sprzętowych. Część z opisanych w pracy funkcjonalności, w szczególności dotyczących operacji wejścia-wyjścia, jest specyficzna dla wersji systemu dla mikrokontrolera LPC1769.

## 2.7. Podsumowanie

Mikrojądro FreeRTOS udostępnia podstawowe mechanizmy do obsługi wielozadaniowości, komunikacji międzyprocesowej, zarządzania dostępem do zasobów współdzielonych oraz do zarządzania pamięcią. Są one jednak wystarczające do wykorzystania w implementacji maszyny wirtualnej Erlanga przeznaczonej dla systemu FreeRTOS.

Część z wymienionych wyżej funkcjonalności FreeRTOS jest na tyle zgodna z oryginalną maszyną wirtualną, że może zostać wykorzystana wprost do implementacji pewnych elementów maszyny. Część z nich musiała jednak ulec częściowej modyfikacji lub zostać całkowicie zaimplementowana, jak np. kolejki wiadomości procesów.

### **3. Język programowania Erlang**



## 4. Zaimplementowane elementy maszyny wirtualnej

Maszyna wirtualna jest warstwą abstrakcji uruchamianą pod kontrolą pewnego systemu operacyjnego. Powinna ona emulować fizyczny procesor w taki sam sposób niezależnie od systemu operacyjnego czy fizycznej architektury na jakiej została uruchomiona. Dzięki temu możliwe jest uruchomienie tego samego kodu, nazywanego kodem pośrednim, przystosowanego do architektury maszyny wirtualnej, pod kontrolą różnych systemów operacyjnych i na różnych architekturach sprzętowych.

Wśród funkcjonalności wirtualnego procesa, które powinna implementować maszyna wirtualna dowolnego języka można wymienić:

- struktury danych, przy pomocy których opisane są instrukcje i ich argumenty;
- stos, wykorzystywany do wywołań funkcji;
- wskaźnik kolejnej instrukcji do wykonania;
- interpreter kodu pośredniego, pobierający kolejną instrukcję do wykonania, dekodujący jej argumenty i wykonujący ją.

W rozdziale wymieniono elementy maszyny wirtualnej Erlanga, które zaimplementowano w pracy, tak aby powyższy zbiór funkcjonalności zapewniał możliwość wykonywania kodu modułów skompilowanych przy użyciu kompilatora Erlanga w wersji R16. Opis poszczególnych elementów zawiera wyjaśnienie ich sposobu działania, ich roli w maszynie wirtualnej, a także porównania funkcjonalności do maszyny wirtualnej BEAM.

Wszystkie opisywane funkcjonalności zostały zaimplementowane w języku C, podobnie jak jest to w przypadku maszyny wirtualnej BEAM.

### 4.1. Moduł ładujący kod (*loader*)

Moduł opisany w niniejszym podrozdziale został zaimplementowany w pliku źródłowym `beam_load.c`.

Podstawowym zadaniem *loadera* jest wykonanie zestawu operacji po których będzie możliwe wykonanie kodu programu, zawartego w pliku będącym efektem kompilacji modułu w języku Erlang, z poziomu interpretera kodu pośredniego w maszynie wirtualnej. W maszynie BEAM źródłem plików binarnych może być system plików systemu operacyjnego lub inny węzeł Erlanga znajdujący się w tym

samym klastrze. Aby pliki te mogły być wykonywane na maszynie zaimplementowanej w ramach pracy, która nie obsługuje ani systemu plików ani protokołu klastrowania, muszą one ulec przetworzeniu i zostać wkompileowane w kod maszyny wirtualnej. Dokonywane jest to za pomocą narzędzia opisanego w dodatku B.

Zawartość pliku, który poddawany jest przetwarzaniu w module została szczegółowo opisana w dodatku D, który ze względu na brak oficjalnej dokumentacji powstał na potrzeby niniejszej pracy.

Aby możliwe było wykonywanie kodu pośredniego przez maszynę wirtualną, konieczne jest wykonanie następujących kroków:

- załadowanie lokalnej tablicy atomów (fragment `Atom`) do globalnej tablicy atomów (por. 4.2.1);
- załadowanie lokalnej tablicy eksportowanych funkcji (fragment `ExpT`) do jej globalnego odpowiednika (por. 4.2.2);
- sparsowanie wyrażeń w postaci *External Term Format*, umieszczonych we fragmencie `LitT` i umieszczenie ich w pamięci o globalnym dostępie (globalnej stercie);
- wyszukanie w globalnej tablicy eksportów funkcji zewnętrznych używanych przez moduł (fragment `ImpT`);
- podstawienie wyrażeń rozpoznawanych globalnie (por. 4.3) za wyrażenia lokalne, opisane w podrozdziale E.1 we fragmencie `Code`. Do rozważanych wyrażeń należą: atomy, etykiety lokalnych funkcji, odnośniki do funkcji znajdujących się w innych modułach oraz wyrażenia umieszczone na globalnej stercie;
- podstawienie za numery operacji z sekcji `Code` (opisane w podrozdziale E.2) wskaźników do odpowiedniej sekcji interpretera kodu maszynowego wykonującego daną instrukcję (por. 4.4).

*Loader* maszyny wirtualnej BEAM wykonuje jeszcze jeden bardzo istotny krok, którego nie wykonuje maszyna zaimplementowana w pracy. Jest nim zastosowanie gramatyki modyfikującej w bardzo istotny sposób kod maszynowy zawarty w pliku binarnym. Gramatykę tę można znaleźć w pliku `ops.tab` w kodzie źródłowym maszyny BEAM. Wynikowy zestaw instrukcji, odpowiadający instrukcjom faktycznie interpretowanym przez BEAM, jest dużo bardziej obszerny od zestawu który może zostać wygenerowany przez kompilator. Motywacją do zmiany kodu maszynowego w ten sposób jest m.in. optymalizacja czasu wykonania często występujących po sobie operacji. W przeciwieństwie do oryginalnej maszyny wirtualnej, interpreter zawarty w niniejszej maszynie wirtualnej dokonuje bezpośredniej interpretacji opkodów, które znajdują się w pliku binarnym z kodem pośrednim.

## 4.2. Tablice

Tablice opisane w tym podrozdziale zostały zaimplementowane w plikach `hash.c`, `index.c`, `atom.c` oraz `export.c`.

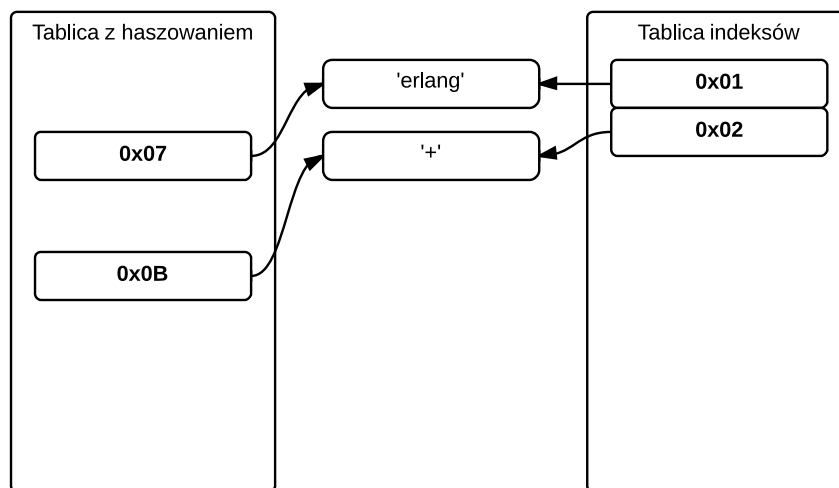


Biorąc pod uwagę modułowy charakter aplikacji napisanych w języku Erlang, maszyna wirtualna musi posiadać pewien mechanizm pozwalający na utrzymywanie globalnego stanu systemu w zależności od aktualnie załadowanych modułów.

Strukturą danych przeznaczoną do tego celu jest tablica z haszowaniem wspomagana przez tablicę indeksów. Połączenie tych dwóch struktur umożliwia wstawienie nowego elementu oraz sprawdzenie jego indeksu w czasie stałym (konieczne jest wyliczenie skrótu). W takim samym czasie (nie jest do tego jednak konieczne wyliczanie skrótu) możliwe jest znalezienie obiektu znając jego indeks. Co więcej, w reprezentacji kodu maszynowego posługiwanie się indeksami obiektów trywializuje ich porównywanie czy pobieranie ich wartości w trakcie jego wykonywania.

Wybór takiej struktury danych ma więc charakter optymalizacyjny. W maszynie wirtualnej tablicowanymi obiektami są: atomy i eksportowane funkcje. Maszyna wirtualna BEAM dodatkowo tablicuje również moduły, gdzie przechowywane są informacje o wskaźnikach do początku kodu modułu w dwóch wersjach: nowej i starej, które mogą działać w maszynie wirtualnej niezależnie od siebie. Ponieważ jednak maszyna rozważana w pracy w obecnej fazie rozwoju nie zapewnia możliwości dynamicznego ładowania modułów, implementacja tej struktury nie było konieczne.

Na rysunku 4.1 przedstawiono sposób w jaki wewnątrz maszyny wirtualnej przechowywane są stabilicowane dane. Przykład ten dotyczy dwuelementowej tablicy atomów, które były do niej wstawiane w kolejności: `erlang`, `+`. Strzałki na diagramie reprezentują przechowywanie wskaźników na struktury atomów przez tablice: z haszowaniem i indeksów.



Rysunek 4.1: Przykład przechowywania danych stabilicowanych danych wewnątrz maszyny wirtualnej.

W maszynie BEAM, w przypadku uruchomionych dużych systemów, powyższe struktury danych mogą zawierać bardzo dużą liczbę elementów, nawet rzędu kilkudziesięciu tysięcy. Zupełnie inaczej sytuacja wygląda w niniejszej maszynie wirtualnej ze względu na jej przeznaczenie, którym są systemy wbudowane i wynikające z tego restrykcyjne limity dostępnej pamięci RAM. Rozmiary tablic nie powinny zatem przekraczać liczby elementów wyrażonej w setkach atomów czy funkcji eksportowanych.

Niemniej jednak, ze względu na możliwość uruchomienia systemu FreeRTOS na mikrokontrolerach o różnych parametrach, pozostawiono możliwość zdefiniowania maksymalnej liczby elementów, jakie mogą zostać wstawione do tablic. Zaimplementowano również mechanizmy automatycznego rozszerzania tablic wraz ze wzrostem liczby elementów, w celu optymalizacji pamięci zajmowanej przez tablice.

Ważną cechą charakterystyczną tablic w maszynie wirtualnej jest również fakt, że raz wstawionego do nich obiektu nie można z niej usunąć. W kontekście maszyny wirtualnej rozważanej w pracy ta cecha nie ma większego znaczenia, gdyż obecnie nie umożliwia ona dynamicznego ładowania modułów po jej uruchomieniu. Jednak w przypadku maszyny BEAM nie należy zapominać o tej cesze np. w sytuacji, gdy program generuje atomy w sposób dynamiczny. Tablica atomów w BEAM może przechowywać aż 1048576 atomów, należy jednak mieć na uwadze to, że próba dodania atomu do pełnej już tablicy zakończy się zakończeniem procesu maszyny wirtualnej.

#### 4.2.1. Tablica atomów

Funkcja skrótu dla atomów (ich reprezentacji w postaci napisu) używana w maszynie wirtualnej to *hashpjw* [5]. Jest to funkcja o bardzo dobrym rozkładzie wartości skrótu dla napisów, jednak wartość zwracana przez oryginalną funkcję jest 32-bitowa.

W celu ograniczenia pamięci zajmowanej przez tablice w maszynie wirtualnej rozważanej w pracy, funkcja haszująca została zmodyfikowana tak, aby zwracała wynik 8-bitowy. Ze względu na duże różnice w rozmiarach tablic pomiędzy rozważaną maszyną a BEAM zmniejszenie długości skrótu zwracanego z funkcji haszującej nie będzie miało wpływu na liczbę kolizji w tablicy z haszowaniem.

Źródłem atomów w tablicy są atomy zdefiniowane w samej maszynie wirtualnej oraz atomy pochodzące z ładowanych modułów. Atomy zdefiniowane ładowane są do tablicy w trakcie uruchamiania maszyny wirtualnej w określonej kolejności, co za tym idzie atomy te mają z góry ustalony i znany indeks, co jest wykorzystywane np. przy definicji funkcji wbudowanych w maszynę wirtualną (por. 4.7). Z kolei indeksy atomów, które pochodzą z ładowanych modułów, a nie zostały wcześniej zdefiniowane, przydzielane są w kolejności ładowania modułów i występowania atomów w tablicach atomów modułów. Równość dwóch atomów oznacza zawsze równość ich indeksów w globalnej tablicy atomów i na odwrót, niezależnie od źródła ich pochodzenia ani momentu załadowania modułu do maszyny wirtualnej.

#### 4.2.2. Tablica eksportowanych funkcji

Funkcja skrótu dla eksportowanych funkcji ma wartość  $M \cdot F + A$ , gdzie  $M$  to indeks w tablicy atomów dla nazwy modułu z którego eksportowana jest funkcja,  $F$  to indeks w tablicy atomów dla nazwy eksportowanej funkcji, a  $A$  to arność tej funkcji.

Wpisy w tablicy eksportowanych funkcji pochodzą z modułów załadowanych do maszyny wirtualnej, dla funkcji które zostały zdefiniowane w lokalnej tablicy eksportów dla danego modułu. W takiej sytuacji w tablicy eksportów przechowywany jest wskaźnik na miejsce w pamięci, w którym znajduje się pierwsza instrukcja funkcji. Interpreter, wykonując kod używa indeksu do odczytania adresu tej instruk-

cji, a następnie wykonuje skok do tego miejsca pamięci i kontynuuje wykonywanie kodu, po uprzednim zapisaniu adresu powrotu.

Elementy tablicy mogą pochodzić również z wbudowanych funkcji (por. 4.7). W tym przypadku, tablica eksportów zawiera wskaźnik do funkcji w języku C, zaimplementowanej jako część maszyny wirtualnej, która zostanie wykonana przez interpreter.

Ponieważ równość indeksów w tablicy eksportów jest równoważna z równością trójek {moduł, funkcja, arność}, w sytuacji dynamicznej podmiany kodu nie jest konieczna zmiana indeksu w załadowanym do pamięci kodzie maszynowym, a tylko odpowiednia zmiana struktury znajdującej się pod tym indeksem.

### 4.3. Typy danych

Erlang jest językiem programowania o dynamicznym, lecz silnym typowaniu. Oznacza to, że każda zmienna, po przypisaniu do niej wartości ma ustalony konkretny typ danych, którego nie można zmienić. Niemożliwe jest również rzutowanie zmiennej na inny typ danych - konwersja do innego typu musi zostać wykonana jawnie a nowa zmienna zajmuje w takiej sytuacji inne miejsce w pamięci programu.

Wszystkie wyrażenia rozpoznawane przez interpreter kodu maszynowego Erlanga zapisane są w postaci wyrażenia takiego samego typu, z punktu widzenia języka C, o długości równej słowu maszynowemu dla danej architektury. W celu rozróżnienia typów zmiennych w pamięci programu, w maszynie wirtualnej wprowadzony został mechanizm **tagowania**, czyli oznaczania w różny sposób zmiennych w pamięci, w zależności od ich typu. Mechanizm ten został zaprojektowany w taki sposób, aby dodatkowy rozmiar w pamięci przeznaczony dla typu był jak najmniejszy. Sposób jego działania został przedstawiony w niniejszym podrozdziale.

#### 4.3.1. Wartości bezpośrednie a pośrednie

Podstawowy podział typów danych wewnątrz maszyny wirtualnej Erlanga wynika ze względu na sposób dostępu do danych.

Jeżeli dana może zostać przechowana na odpowiednio małym obszarze pamięci, czyli w jednym słowie maszynowym (w przypadku niniejszej maszyny są to 32 bity) z uwzględnieniem tagu oznaczającego typ to dane tego typu nazywane są wartościami bezpośrednimi (**IMMED**). Aby dokonać tagowania lub odczytania wartości ze zmiennej tego typu wystarczy wykonać jedną operację przesunięcia bitowego.

Przeciwieństwem danych bezpośrednich są dane pośrednie, które mogą przybrać postać listy (**CONS**) lub typu opakowanego (**BOXED**). Wyrażenia oznaczone tagiem dla jednego z tych typów przechowują fizyczny wskaźnik na miejsce w pamięci, gdzie znajdują się dla nich właściwe dane. Przy tagowaniu wskaźników wykorzystany został fakt, że bloki pamięci alokowane przez maszynę wirtualną są zawsze wielokrotnością całego słowa maszynowego. Co za tym idzie dwa najmniej znaczące bity wskaźnika, na maszynie uruchomionej w architekturze 32-bitowej lub wyższej będą zawsze zerami co

można wykorzystać do przechowania dodatkowej, dwubitowej, informacji. W tym wypadku jest to tag rozróżniający wskaźniki na listy od wskaźników na typy opakowane oraz od pozostałych wyrażeń.

Tabela 4.1 prezentuje sposób tagowania ww. typów danych. Tagi dla poszczególnych typów danych zostały w zapisie słowa maszynowego pogrubione.

Na przykład do przechowania wskaźnika do pierwszego elementu listy, znajdującego się pod adresem 128, w pamięci zapisane zostanie wyrażenie:

$$10000000 \vee \mathbf{01} = 00000000 \ 00000000 \ 00000000 \ 100000\mathbf{01}$$

Do odczytania wartości wskaźnika wystarczy więc wyzerowanie dwóch najmniej znaczących bitów wyrażenia:

$$00000000 \ 00000000 \ 00000000 \ 100000\mathbf{01} \wedge \neg(11) = 10000000$$

Typ danych	Słowo maszynowe (binarne)	Opis
<b>IMMED</b>	IIIIIIII IIIIIIII IIIIIIII IIBBTT <b>11</b>	<ul style="list-style-type: none"> <li>– bity T budują konkretny tag (por. 4.3.2) typu bezpośredniego;</li> <li>– bity I oznaczają wartość przechowywaną przez wyrażenie, która w zależności od rozmiaru tagu może mieć 26 lub 28 bitów;</li> <li>– bity B mogą być dwoma najbardziej znaczącymi bitami tagu lub dwoma najmniej znaczącymi bitami przechowywanej wartości, w zależności od typu.</li> </ul>
<b>CONS</b>	PPPPPPPP PPPPPPPP PPPPPPPP PPPPPP <b>01</b>	<ul style="list-style-type: none"> <li>– bity P są 30 najbardziej znaczącymi bitami wskaźnika do wyrażenia stanowiącego pierwszy element listy, dwa najmniej znaczące bity zawsze będą zerami dlatego mogą zostać nadpisane przez tag.</li> </ul>
<b>BOXED</b>	PPPPPPPP PPPPPPPP PPPPPPPP PPPPPP <b>10</b>	<ul style="list-style-type: none"> <li>– bity P są 30 najbardziej znaczącymi bitami wskaźnika do nagłówka identyfikujące typ i rozmiar opakowanych danych, w tym przypadku również dwa najmniej znaczące bity zawsze będą zerami.</li> </ul>

Tablica 4.1: Rozróżnienie tagów ze względu na sposób dostępu do danych

### 4.3.2. Wartości bezpośrednie

Wartości bezpośrednie (**IMMED**) mogą być przechowywane przez różne typy danych, dla których przewidziano dodatkowe 2 lub 4 bity na tag. Tabela 4.2 podsumowuje wszystkie zaimplementowane w maszynie wirtualnej opisywanej w pracy typy zawierające wartości bezpośrednie.

Typ danych	Słowo maszynowe (binarnie)	Opis
<b>PID</b>	IIIIIIII IIIIIIII IIIIIIII II110011	Wartością przechowywaną przez wyrażenie tego typu jest indeks procesu w tablicy procesów w maszynie wirtualnej (por. 4.5).
<b>SMALL_INT</b>	IIIIIIII IIIIIIII IIIIIIII II1111	Przechowywaną wartością jest liczba całkowita (ze znakiem), którą można zapisać na maksymalnie 28 bitach w pamięci.
<b>ATOM</b>	IIIIIIII IIIIIIII IIIIIIII II001011	Wartością przechowywaną jest indeks atomu w tablicy atomów (por. 4.2.1). Dzięki takiemu zapisowi porównanie dwóch dowolnych atomów sprowadza się do porównania dwóch 32-bitowych liczb.
<b>NIL</b>	00000000 00000000 00000000 00111011	Przechowywaną wartością jest zawsze zero. Wyrażenie to służy do oznaczania końca listy.

Tablica 4.2: Rozróżnienie tagów dla danych bezpośrednich

Na przykład atom mający indeks  $2_{10} = 10_2$  w tablicy atomów w pamięci będzie zapisany w postaci:

$$(10 \ll 6) \vee 001011 = 00000000 \ 00000000 \ 00000000 \ 10001011$$

Do odczytania indeksu atomu wystarczy zatem wykonać operację przesunięcia bitowego w prawo:

$$00000000 \ 00000000 \ 00000000 \ 10001011 \gg 6 = 10$$

### 4.3.3. Listy

Listy są jednym ze złożonych typów danych obsługiwanych przez język Erlang. W maszynie wirtualnej zaimplementowane zostały przy użyciu listy jednokierunkowej. Wyrażenie, które służy np. do przechowywania listy na stosie procesu lub przekazywania jej jako argument do funkcji otagowane jest jako typ **CONS** (por. 4.3.1) i zawiera wskaźnik do pierwszego elementu listy. Element listy jest zwykłym wyrażeniem erlangowym, a więc zajmuje jedno słowo maszynowe. Słowem maszynowym następującym po elemencie jest kolejne wyrażenie typu **CONS**, które zawiera wskaźnik do kolejnego elementu listy. Wyrażenie to może być również typu **NIL** (por. 4.3.2), co oznacza że dany element był ostatnim elementem listy.

W Erlangu nie ma osobnego typu do przechowywania ciągu znaków. Udostępniony lukier składniowy pozwala jednak na posługiwanie się napisami, np. w postaci: "hello". Wyrażenie tego typu

zostanie jednak zinterpretowane jako lista liczb całkowitych, odpowiadającymi kodom ASCII kolejnych liter w napisie. W tym przypadku będzie to następująca lista: `[104, 101, 108, 108, 111]`.

Na rysunku 4.2 zaprezentowano przykładową stertę procesu Erlanga zawierającą powyższą listę, wraz z wyjaśnieniem typu i wartości zawartych w poszczególnych słowach maszynowych. Wyrażenie będące początkiem listy znajduje się w pierwszym wierszu sterty. Strzałki na diagramie reprezentują zawieranie wskaźnika do innego miejsca w pamięci przez wyrażenie z którego wychodzą.

Adres	Słowo maszynowe	Opis wyrażenia
	00000000 00000000 00000000 10000001	128 CONS
132	00000000 00000000 00000000 01111001	120 CONS
128	00000000 00000000 00000110 10001111	104 SMALL_INT
124	00000000 00000000 00000000 01110001	112 CONS
120	00000000 00000000 00000110 01011111	101 SMALL_INT
116	00000000 00000000 00000000 01101001	104 CONS
112	00000000 00000000 00000110 11001111	108 SMALL_INT
108	00000000 00000000 00000000 01100001	096 CONS
104	00000000 00000000 00000110 11001111	108 SMALL_INT
100	00000000 00000000 00000000 00111011	NIL
96	00000000 00000000 00000110 11111111	111 SMALL_INT

Rysunek 4.2: Przykład przechowywania listy na sterce procesu

Powyższy przykład dobrze ilustruje narzut pamięciowy jaki wprowadza sposób zapisu napisu przy użyciu listy. Informacja, która przy użyciu innych języków programowania może być zapisana przy użyciu 5 bajtów w języku Erlang potrzebuje aż 10 słów maszynowych (40 bajtów na architekturze 32-bitowej). Receptą na tego typu problem, wprowadzoną w maszynie wirtualnej BEAM, jest binarny typ danych. Napis "hello" przy jego użyciu zajmowałby w pamięci 3 słowa maszynowe (nagłówek i 2 słowa przeznaczone na dane). Typ ten jednak nie został zaimplementowany w obecnej wersji maszyny na system FreeRTOS.

Jak można zauważyć zarówno złożoność obliczeniowa (czas dostępu do danych na liście jest liniowy) jak i pamięciowa przy wykorzystaniu tego typu danych jest dość znacząca.

#### 4.3.4. Krotki

Kolejnym złożonym typem danym, z którego można korzystać w języku Erlang jest krotka, zajmująca spójny obszar pamięci. Z implementacyjnego punktu widzenia można porównać ją do tablicy zawierającej wyrażenia erlangowe.

Krotka jest jednym z typów opakowanych (**BOXED**), zatem referencja do niej z poziomu stosu procesu zawiera wskaźnik do nagłówka krotki. Nagłówek, podobnie jak pozostałe wyrażenia zajmuje jedno słowo maszynowe i przechowuje rozmiar krotki w postaci:

AAAAAAAA AAAAAAAAA AAAAAAAAA AA000000

gdzie bity A oznaczają rozmiar (arność) krotki. Wyrażenia wchodzące w skład krotki zajmują kolejne, następujące po nagłówku, słowa maszynowe.

Na rysunku 4.3 zaprezentowany został przykład przechowywania danych wewnątrz krotki na sterzie procesu, jak w przykładzie na rys. 4.2, czyli krotki {104, 101, 108, 108, 111}.

Adres	Słowo maszynowe	Opis wyrażenia
	00000000 00000000 00000000 10000110	132 BOXED
132	00000000 00000000 00000001 01000000	005 ARITYVAL
128	00000000 00000000 00000110 10001111	104 SMALL_INT
124	00000000 00000000 00000110 01011111	101 SMALL_INT
120	00000000 00000000 00000110 11001111	108 SMALL_INT
116	00000000 00000000 00000110 11001111	108 SMALL_INT
112	00000000 00000000 00000110 11111111	111 SMALL_INT

Rysunek 4.3: Przykład przechowywania krotki na sterze procesu

Ze względu na to, że krotka przechowuje dane w spójnym obszarze pamięci i dostęp do nich odbywa się w czasie stałym, użycie tego typu danych jest dobrym pomysłem w przypadku przechowywania danych tablicowych.

#### 4.3.5. Duże liczby

Drugim zaimplementowanym, opakowanym typem danych są duże liczby całkowite. Wszystkie liczby całkowite, które nie mieszczą się w zakresie typu **SMALL\_INT**, czyli potrzebują do ich zapisania przynajmniej 29 bitów zapisywane są w typie danych **BIGNUM**. Maszyna wirtualna zaimplementowana w ramach pracy, podobnie jak maszyna BEAM, implementuje własną arytmetykę implementującą operacje na tego typu liczbach.


Nagłówek typu **BOXED** ma w tym przypadku postać:

AAAAAAAA AAAAAAAAA AAAAAAAAA AA001S00

gdzie bity A oznaczają liczbę słów maszynowych, które składają się na całą liczbę bez znaku. Słowa te, w kolejności od najmniej do najbardziej znaczącego, zajmują w pamięci kolejne słowa maszynowe po nagłówku. Bit S przechowuje znak liczby: 1 dla liczb ujemnych, 0 dla dodatnich.

Na rysunku 4.4 zaprezentowany został przykład przechowania liczby  $10^{28}$  na sterze procesu.

Adres	Słowo maszynowe	Opis wyrażenia
	00000000 00000000 00000000 10000110	132 BOXED
132	00000000 00000000 00000000 11001000	003 BIGNUM (+)
128	00010000 00000000 00000000 00000000	0x10000000
124	00111110 00100101 00000010 01100001	0x3e250261
120	00100000 01001111 11001110 01011110	0x204fce5e



Rysunek 4.4: Przykład przechowywania dużej liczby na sterce procesu

#### 4.3.6. Niezaimplementowane typy danych

Typy danych dostępne w maszynie BEAM, które nie zostały zaimplementowane w pracy to:

- **lambdy** (tag **FUN**) - ten typ danych używany jest w komunikacji między węzłami wewnątrz klastra Erlanga, co nie jest aktualnie obsługiwane przez maszynę wirtualną. Lambdy pojawiające się w kodzie źródłowym modułów reprezentowane są za pomocą funkcji lokalnych i nie potrzebują osobnego typu danych do poprawnego działania;
- **referencje** (tag **REF**) - referencje z założenia używane są do oznaczania wiadomości wysyłanych do innych węzłów z uruchomioną maszyną wirtualną a klastrowanie nie jest wspierane przez implementowaną maszynę wirtualną;
- **porty** (tag **PORT**) - porty używane są do identyfikacji procesów uruchomionych w systemie operacyjnym poza maszyną wirtualną Erlanga, a do których delegowane są pewne operacje, wykonywane na poziomie systemu operacyjnego. Wykonanie tych operacji (jak np. operacje na plikach) z założenia może zająć pewien dłuższy okres czasu, co mogłoby zakłócić harmonogramowanie procesów. Typ danych nie został zaimplementowany, gdyż w maszynie wirtualnej wszystkie operacje wykonywane na poziomie mikrojądra zostały zaimplementowane przy pomocy funkcji wbudowanych (por. 4.7);
- **liczby zmiennoprzecinkowe** (tag **FLONUM**) - ten typ danych również nie został zaimplementowany w wersji maszyny opisywanej w pracy. Liczby zmiennoprzecinkowe rzadko wykorzystywane są w programowaniu urządzeń wbudowanych ze względu na bardzo duży narzut czasowych w przypadku mikrokontrolerów nie mających koprocatora (np. procesor ARM Cortex-M3 nie posiada FPU). M.in. z tego powodu do maszyny BEAM arytmetyka zmiennoprzecinkowa została dodana dopiero w wersji R8;
- **binaria** (tagi \*\_**BINARY**) - binarny typ danych włącznie ze wszystkimi operacjami dotyczącymi dopasowywania do nich zmiennych również nie został zaimplementowany w maszynie. Jest to funkcjonalność warta rozważenia w przypadku dalszej pracy nad maszyną ze względu na binarny



charakter szeregowych interfejsów wejścia-wyjścia obsługiwanych przez mikrokontrolery. Podstawowe operacje na binariach do maszyny BEAM zostały dodane w wersji R7, bardziej zaawansowane były sukcesywnie dodawane od wersji R10;

- **etykiety bloku `catch`** (tag **CATCH**) - typ danych służy do zapamiętywania na stosie procesu miejsc w kodzie, w których pojawiają się bloki `try` oraz `catch`. W przypadku błędu w programie, przed zakończeniem działania procesu, stos jest przeszukiwany w poszukiwaniu obecności ww. bloków. Instrukcje dotyczące łapania błędów nie zostały zaimplementowane w niniejszej maszynie wirtualnej.

## 4.4. Interpreter kodu maszynowego

Moduł opisany w niniejszym podrozdziale został zaimplementowany w pliku źródłowym `beam_emu.c`.

### 4.4.1. Maszyna stosowa a rejestrowa

Spośród sposobów implementacji maszyny wirtualnej można wymienić dwa: maszynę stosową i rejestrową. Różnicę między tymi dwoma podejściami stanowi sposób przechowywania argumentów wywoływanej funkcji, miejsca zapisu wyniku jej wykonania oraz zmiennych tymczasowych przez nią używanych.

W przypadku maszyny stosowej dane te przechowywane są na stosie. Kolejne argumenty operacji umieszczane są na stosie za pomocą operacji **PUSH**, natomiast przed wykonaniem operacji zdejmowane są przez instrukcję **POP**. Instrukcje wykonywane przez maszynę wirtualną nie potrzebują zatem żadnych dodatkowych argumentów, gdyż te powinny być przed jej wywołaniem umieszczone na szczycie stosu, podobnie jak wynik zwrócony przez wykonaną instrukcję.

Z kolei w przypadku maszyny rejestrowej, powyższe informacje zapisywane są w zbiorze rejestrów. Konsekwencją tego jest brak instrukcji manipulujących stosem w wykonywanym kodzie, co wpływa pozytywnie na szybkość działania interpretera kodu pośredniego. Właściwe instrukcje wymagają jednak przekazania dodatkowych argumentów adresujących rejestry, w których znajdują się argumenty operacji i rejestr docelowy dla wyniku jej wykonania. Efektem tego jest dłuższy zapis instrukcji w kodzie pośrednim niż w przypadku maszyny stosowej. Dodatkowym atutem użycia rejestrów jest możliwość optymalizacji kodu polegającego na wyliczeniu i zapisaniu do rejestru pewnego pośredniego wyniku. Wynik ten może następnie zostać wykorzystany przez kilka różnych instrukcji, zamiast wyliczania go od nowa przez każdą z nich.

Przykładami stosowych maszyn są maszyny języków takich jak Java czy .NET. Z kolei przykładami maszyn rejestrowych są maszyny wirtualne języka Lua czy maszyna wirtualna Javy na system Android - Dalvik.

#### 4.4.2. Model interpretera

Maszyna wirtualna języka Erlang (BEAM oraz maszyna zaimplementowana w pracy) jest również przykładem rejestrowej maszyny wirtualnej.

Interpreter korzysta z następujących rejestrów:

- rejestry **X<sub>0</sub>-X<sub>255</sub>** służące do przechowywania kolejnych argumentów z jakimi wywoływana jest funkcja, dodatkowo w rejestrze **X<sub>0</sub>** zapisywana jest wartość zwracana przez funkcję;
- rejestry **Y** znajdujące się na stosie aktualnie uruchomionego procesu, służące do przechowywania zmiennych lokalnych;
- rejestr **IP** przechowujący wskaźnik do aktualnie wykonywanej przez interpreter instrukcji;
- rejestr **CP** przechowujący adres powrotu - wskaźnik do instrukcji, którą interpreter powinien wykonać gdy nastąpi powrót z aktualnie wykonywanej funkcji.

Na listingu 4.1 zaprezentowany został przykładowy kod pośredni funkcji o arności 3, wyliczającej maksimum ze wszystkich jej argumentów, wykorzystujący do tego celu funkcję wbudowaną `erlang:max/2`. Wyjaśnienie działania poszczególnych operacji zostało zawarte w dodatku E.

```

1 | {allocate, 1, 3}.
2 | {move, {x, 2}, {y, 0}}.
3 | {call_ext, 2, {extfunc, erlang, max, 2}}.
4 | {move, {y, 0}, {x, 1}}.
5 | {call_ext_last, 2, {extfunc, erlang, max, 2}, 1}.

```

Listing 4.1: Kod pośredni funkcji zwracającej maksimum z trzech argumentów

Na rysunkach od 4.5 do 4.12 zaprezentowano zawartość rejestrów X (interpretera) i Y (stos procesu) po wykonaniu poszczególnych operacji kodu powyższego dla argumentów: 5, 7, 9. Zapis {1, 1} oznacza, że wskaźnik do instrukcji wskazuje na linię 1 z przykładu na listingu 4.1, z kolei `erlang:max/2` oznacza, że wskaźnik do instrukcji wskazuje na początek tej funkcji wbudowanej. Przez zapis CP rozumiana jest wartość wskaźnika CP w chwili wywołania funkcji.

Jak można zauważyć na rys. 4.5 wszystkie trzy argumenty zostały umieszczone w rejestrach X: 0, 1 i 2. Instrukcja `{allocate, 1, 3}`, znajdująca się w pierwszej linii powoduje rozszerzenie stosu procesu o 2 wyrażenia, z czego na szczycie stosu umieszczany jest adres powrotu, z jakim została wywołana funkcja, zapisany pod wskaźnikiem CP. Jeżeli w trakcie wykonania tej instrukcji konieczne byłoby uruchomienie *garbage collector*a, drugi argument informuje o tym, że aktualnie w użyciu są 3 rejestry X i nie jest możliwe zwolnienie obszarów pamięci do których one się odnoszą.

W linii 2 (rys. 4.7) dokonane zostaje przeniesienie wyrażenia z rejestru X<sub>2</sub> do rejestru Y<sub>0</sub>, który jest pierwszym wyrażeniem na stosie poniżej jego szczytu. Należy zauważyć, że pomimo wykorzystania stosu do zapisu zmiennych lokalnych, które nie zostaną zmodyfikowane pomiędzy wywołaniami funkcji,

do operacji na nim nie wykorzystuje się operacji stosowych. Nie należy zatem wiązać wykorzystania stosu procesu z faktem, że maszyna wirtualna jest maszyną stosową.

Przed wywołaniem funkcji `erlang:max/2` w linii 3, jako adres powrotu zostaje zapisana kolejna instrukcja w module, znajdująca się w linii 4. Argumentami wywoływanej funkcji są wartości znajdujące się w rejestrach  $X_0$  i  $X_1$ . Wywołana funkcja w momencie powrotu przepisuje wskaźnik powrotu (**CP**) na kolejną instrukcję do wykonania (**IP**). Wartość zwrócona z funkcji znajduje się w rejestrze  $X_0$ .

W linii 5 (rys. 4.10) wartość przechowywana na stosie zostaje przepisana do rejestru  $X_1$ . Drugie wywołanie funkcji `erlang:max/2` jest wywołaniem ogonowo-rekurencyjnym. Dlatego też przykładowa funkcja odpowiedzialna jest za przywrócenie wskaźnika **CP** ze stosu i zwolnienie go jeszcze przed wywołaniem funkcji zewnętrznej. Wartość zwrócona z trójargumentowej funkcji znajduje się w rejestrze  $X_0$ , a kolejna wykonana instrukcja będzie następująca po instrukcji, która ją wywołała.

#### 4.4.3. Sposób implementacji

Z implementacyjnego punktu widzenia kod interpretera kodu pośredniego w maszynie zaimplementowanej w pracy interpretuje kod adresowany bezpośrednio (*Directly Threaded Code* [10]). Oznacza to, że operacja przekazywana do interpretera zapisana jest nie w postaci opkodu, ale zawiera wskaźnik do etykiety implementującej daną instrukcję. Odpowiedniej podmiany, w momencie ładowania pliku z modulem dokonuje *loader*.

Maszyna wirtualna BEAM implementuje także sposób adresowania operacji przez ich opkod, na wypadek gdyby kompilator języka C użyty do jej kompilacji nie obsługiwał rozszerzenia standardu GNU C, jakim są wskaźniki do etykiet. Metoda *Switch Threading* nie wymaga podmiany w żaden sposób oryginalnych opkodów, jednak interpretacja kodu pośredniego z jej użyciem jest dużo wolniejsza. Powodem tego jest fakt, że kod każdej operacji przed jej wykonaniem należy sprawdzić z całym zestawem obsługiwanych opkodów w instrukcji *switch*, aż do napotkania właściwej. W maszynie na system FreeRTOS wykorzystano tylko adresowanie bezpośrednie, gdyż kompilator **gcc** wykorzystywany do jej kompilacji posiada ww. funkcjonalność.

Zestaw instrukcji zaimplementowanych w maszynie wirtualnej wraz z opisem ich działania został zaprezentowany w dodatku E. Dodatek ten wymienia operacje, jakie możliwe są do otrzymania w pliku z kodem pośrednim z kompilatora języka Erlang. Maszyna wirtualna BEAM implementuje jednak nieco większy zestaw instrukcji, który zostaje podmieniany w kodzie maszynowym w momencie ładowania modułu do pamięci (por. 4.1).

Dodatkowymi operacjami, wykorzystywanymi wewnętrznie przez maszynę wirtualną, zaimplementowanymi w interpreterze są: `beam_apply/0` oraz `normal_exit/0`, służące do kontroli cyklu życia procesu. W momencie startu procesu jako pierwsza instrukcja do wykonania (wskaźnik **IP**) zostaje ustawiona pierwsza z tych operacji, po wcześniejszym zapisaniu do pierwszych trzech rejestrów  $X$ : modułu, funkcji i listy argumentów będących początkowymi dla tego procesu. Następnie operacja ta wywołuje właśnie tę funkcję. Jako adres powrotu z wywołania funkcji początkowej (wskaźnik **CP**) ustawiana jest druga z dodatkowych instrukcji, która kończy działanie procesu z powodu osiągnięcia końca jego kodu.

Rejestry (X)		Stos (Y)	Instrukcje	
{x, 0}	5		IP	{1, 1}
{x, 1}	7		CP	CP
{x, 2}	9			

Rysunek 4.5: Rejestry przed wykonaniem instrukcji w linii 1

Rejestry (X)		Stos (Y)	Instrukcje	
{x, 0}	5	{y, -1} CP	IP	{1, 2}
{x, 1}	7	{y, 0} ?	CP	CP
{x, 2}	9			

Rysunek 4.6: Rejestry przed wykonaniem instrukcji w linii 2

Rejestry (X)		Stos (Y)	Instrukcje	
{x, 0}	5	{y, -1} CP	IP	{1, 3}
{x, 1}	7	{y, 0} 9	CP	CP
{x, 2}	9			

Rysunek 4.7: Rejestry przed wykonaniem instrukcji w linii 3

Rejestry (X)		Stos (Y)	Instrukcje	
{x, 0}	5	{y, -1} CP	IP	erlang: max/2
{x, 1}	7	{y, 0} 9	CP	{1, 4}
{x, 2}	9			

Rysunek 4.8: Rejestry przed wywołaniem funkcji w linii 3

Rejestry (X)		Stos (Y)	Instrukcje	
{x, 0}	7	{y, -1} CP	IP	{1, 4}
{x, 1}	7	{y, 0} 9	CP	{1, 4}
{x, 2}	9			

Rysunek 4.9: Rejestry przed wykonaniem instrukcji w linii 4

Rejestry (X)		Stos (Y)	Instrukcje	
{x, 0}	7	{y, -1} CP	IP	{1, 5}
{x, 1}	9	{y, 0} 9	CP	{1, 4}
{x, 2}	9			

Rysunek 4.10: Rejestry przed wykonaniem instrukcji w linii 5

Rejestry (X)		Stos (Y)	Instrukcje	
{x, 0}	7		IP	erlang: max/2
{x, 1}	9		CP	CP
{x, 2}	9			

Rysunek 4.11: Rejestry przed wywołaniem funkcji w linii 5

Rejestry (X)		Stos (Y)	Instrukcje	
{x, 0}	9		IP	CP
{x, 1}	9		CP	CP
{x, 2}	9			

Rysunek 4.12: Rejestry po wykonaniu instrukcji w linii 5, a zarazem i całej funkcji

## 4.5. Procesy

Logika procesu opisana w niniejszym podrozdziale została zaimplementowana w pliku źródłowym `erl_process.c`.

### 4.5.1. Tablica procesów

Struktura procesów przechowywana jest w prealokowanej tablicy o konfigurowalnym rozmiarze. W zaimplementowanej maszynie domyślnie rozmiar ten wynosi 25. Tak mała liczba procesów została wybrana ze względu na duże ograniczenia zasobów platformy docelowej dla maszyny.

Procesy identyfikowane są wewnątrz maszyny wirtualnej przez identyfikator procesu (**PID**, por. 4.3.2). Ten bezpośredni typ danych przechowuje indeks procesu w ww. tablicy, dzięki czemu możliwe jest szybkie znalezienie struktury procesu, który konieczny jest do wykonania instrukcji przez interpreter kodu.

### 4.5.2. Struktura procesu

W strukturze przechowującej informacje o procesie zostały zawarte dane niezbędne do poprawnego wykonywania przez niego kodu. Spośród nich można wymienić:

- wspomniany wcześniej identyfikator procesu (**PID**);
- blok pamięci zajmowany przez wyrażenia wykorzystywane przez proces, zawierający stertę i stos. Został on szczegółowo opisany w podrozdziale 4.8;
- wskaźniki przechowujące aktualnie wykonywaną przez proces instrukcję oraz adres powrotu;
- liczba redukcji jakie pozostały do wyłączenia procesu;
- uchwyt do zadania w systemie FreeRTOS, w kontekście którego wykonywany jest kod procesu;
- pamięć do przechowania wartości rejestrów **X** w sytuacji gdy proces przestanie być wykonywany na rzecz innego procesu (por. 4.6);
- kolejka wiadomości, które zostały wysłane do procesu;
- lista procesów połączonych z danym procesem poprzez *link* (por. ??);
- flagi procesu, ustawiane przez funkcję `erlang:process_flag/2`;
- informacje statystyczne związane z cyklem życia procesu, takie jak liczba wykonanych redukcji czy liczba wykonanych odświeżeń na procesie w trakcie jego działania.

Podstawową różnicą między implementacją maszyny a maszyny BEAM jest sposób wykonywania kodu pośredniego procesu. Maszyna BEAM implementuje własny algorytm harmonogramowania, biorąc

na siebie odpowiedzialność za utrzymywanie kolejki procesów do wykonania i zarządzania kolejnością ich wykonania. Logika procesu nie jest zatem opakowana w żadną warstwę abstrakcji. Z kolei maszyna zaimplementowana w pracy do zarządzania kolejnością procesów wykorzystuje *scheduler* wbudowany w mikrojądro FreeRTOS, a procesy są instancjami zadań przez niego zarządzanych (por. podrozdział 2.1). Powodem wykorzystania takiego podejścia jest mały narzut pamięciowy i czasowy związany z uruchamianiem zadań, a także wbudowana obsługa priorytetów zadań i możliwość konfiguracji parametrów planisty.

Różnicą w strukturze procesu jest także uproszczenie obszaru pamięci dla wyrażeń procesu do jednej, podstawowej sterty. Maszyna BEAM wykorzystuje bardziej zaawansowane, generacyjne podejście przechowując wyrażenia na dwóch stertach. Technika ta opisana została w podrozdziale 4.8.

W maszynie rozważanej w pracy nie została także zaimplementowany słownik procesu.

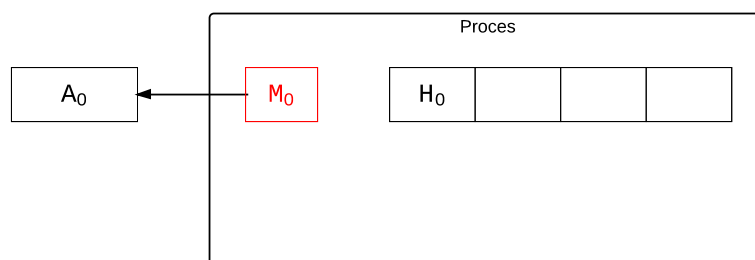
### 4.5.3. Komunikacja międzyprocesowa

Komunikacja między procesami w maszynie wirtualnej jest zapewniona dzięki instrukcjom:

- `send`, służącej do wysłania wiadomości do innego procesu;
- `wait` i `wait_timeout` zawieszających działanie procesu aż do momentu otrzymania wiadomości lub przeterminowania;
- `loop_rec`, `remove_message` i `loop_rec_end`, wykonujących operacje na kolejce wiadomości.

Mechanizm wysyłania wiadomości do procesu został zaprezentowany na rysunkach od 4.13 do 4.16.

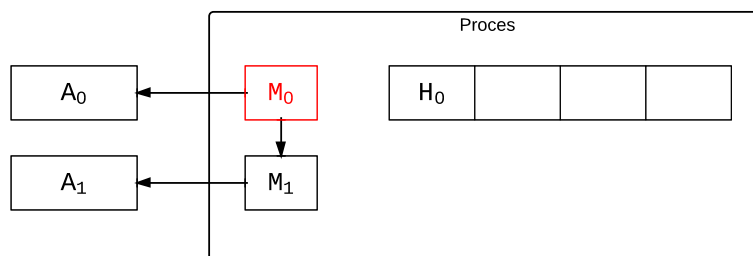
Na początku (rys. 4.13) w kolejce wiadomości procesu odbierającego znajduje się jedna wiadomość ( $M_0$ ). Wskaźnik aktualnej wiadomości ustawiony jest na pierwszą wiadomość w kolejce, oznaczoną kolorem czerwonym. Zostanie on wykorzystany przez proces w momencie osiągnięcia przez kod procesu bloku `receive`.



Rysunek 4.13: Kolejka wiadomości procesu z jedną, nieodczytaną wiadomością.

Wysyłanie wiadomości do procesu polega na dołączeniu na koniec kolejki (która zaimplementowana jest jako lista jednokierunkowa) nowej wiadomości (rys. 4.14). Jeżeli wysyłane wyrażenie nie jest bezpośredniego typu danych, całe wyrażenie jest kopiowane do obszaru pamięci poza pamięcią należącą

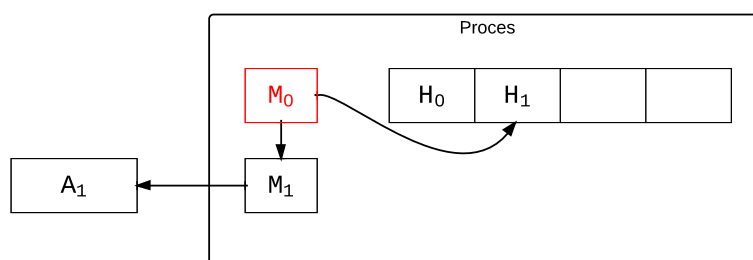
do procesu-odbiorcy. Wiadomość nie jest kopiowana bezpośrednio na stertę procesu, gdyż w sytuacji w której na sterce nie byłoby wystarczającej ilości miejsca, konieczne byłoby uruchomienie *garbage collector*. Tutaj pamięć zajmowana przez treść wiadomości została oznaczona przez  $A_0$  oraz  $A_1$ . Jego uruchomienie powinno jednak zawsze mieć miejsce w bezpiecznym miejscu kodu, w którym znany jest cały źródłowy zbiór wyrażeń.



Rysunek 4.14: Kolejka wiadomości procesu z dodaną kolejną wiadomością.

W momencie, gdy proces-odbiorca w wykonywanym przez siebie kodzie dochodzi do instrukcji `loop_rec` pobierana jest przez niego wiadomość oznaczona wskaźnikiem aktualnej wiadomości. W sytuacji, gdy kolejka jest pusta proces zostaje zawieszany aż do momentu otrzymania kolejnej wiadomości lub zrealizowania się przeterminowania. W pierwszym przypadku proces ponownie próbuje pobrać wiadomość z kolejki, w drugim zaś wykonywany jest kod przewidziany po zejściu przeterminowania.

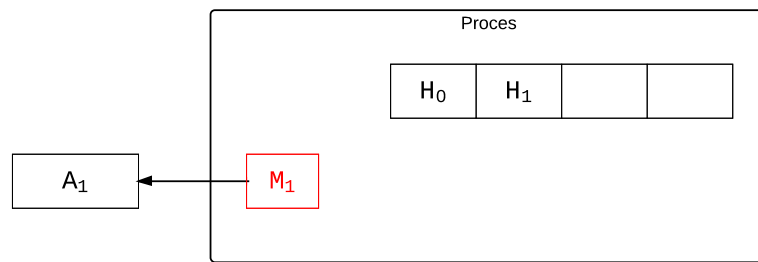
Odebranie wiadomości przez proces polega na skopiowaniu jego zawartości na własną stertę (wyrażenie  $H_1$  na rys. 4.15), zwolnieniu pamięci zajmowanej przez jej oryginalną zawartość, usunięciu wiadomości z kolejki wiadomości oraz przepisaniu wskaźnika aktualnej wiadomości na następną w kolejce (rys. 4.16). Cała procedura jest powtarzana w momencie ponownego dojścia do instrukcji `loop_rec` w kodzie maszynowym (bloku `receive`).



Rysunek 4.15: Kolejka wiadomości procesu z odczytaną pierwszą wiadomością.

#### 4.5.4. Obsługa błędów

Maszyna wirtualna opisywana w niniejszej pracy posiada mechanizm dwukierunkowego połączenia procesów (*link*), który zapewnia propagację błędu, w sytuacji gdy w trakcie wykonywania się jednego z połączonych procesów wystąpi błąd, na skutek którego jego działanie zostanie zakończone. Domyśl-



Rysunek 4.16: Kolejka wiadomości procesu po usunięciu pierwszej wiadomości.

nym zachowaniem procesu otrzymującego informację o zakończeniu działania procesu jest w takiej sytuacji jego zakończenie z takim samym błędem. Możliwe jest jednak ustawienie w nim flagi `trap_exit`, czego efektem jest zamiana ww. sygnału na wiadomość wysłaną do tego procesu.

Maszyna wirtualna BEAM oprócz ww. mechanizmu zapewnia również mechanizm jednokierunkowej obserwacji procesów (*monitor*), a także bloki przechwytywania błędów w kodzie (`try` i `catch`). Funkcjonalności te nie zostały jednak zapewnione w maszynie zaimplementowanej w ramach niniejszej pracy.

Lista błędów, jakie mogą wystąpić w trakcie uruchomienia procesu, wraz z opisem sytuacji w jakiej mogą wystąpić została zawarta w dokumentacji języka [4].

## 4.6. Planista (*scheduler*)

W systemie współbieżnym procesy mogą działać na jeden z dwóch sposobów:

- we współbieżności konkurencyjnej (*pre-emptive*), w której planista decyduje o tym kiedy przerwać wykonywanie pewnego procesu i wznowić inne;
- we współbieżności współpracującej (*cooperative*), w której proces decyduje kiedy przerwać swoje wykonywanie po to, aby *scheduler* mógł wybrać inny proces do wykonania.

Z punktu widzenia programisty Erlanga procesy uruchomione w ramach systemu działają we współbieżności konkurencyjnej, gdyż planista sam zarządza kolejką procesów przeznaczonych do wykonania. Implementacja logiki procesu w maszynie wirtualnej opiera się jednak na współbieżności współpracującej, ponieważ to ona jest odpowiedzialna za wywołanie funkcji wybierającej kolejny proces do wykonania w odpowiedniej chwili. Wynika to z faktu, że wyłączenie aktualnie wykonywanego procesu może nastąpić tylko pod pewnymi warunkami.

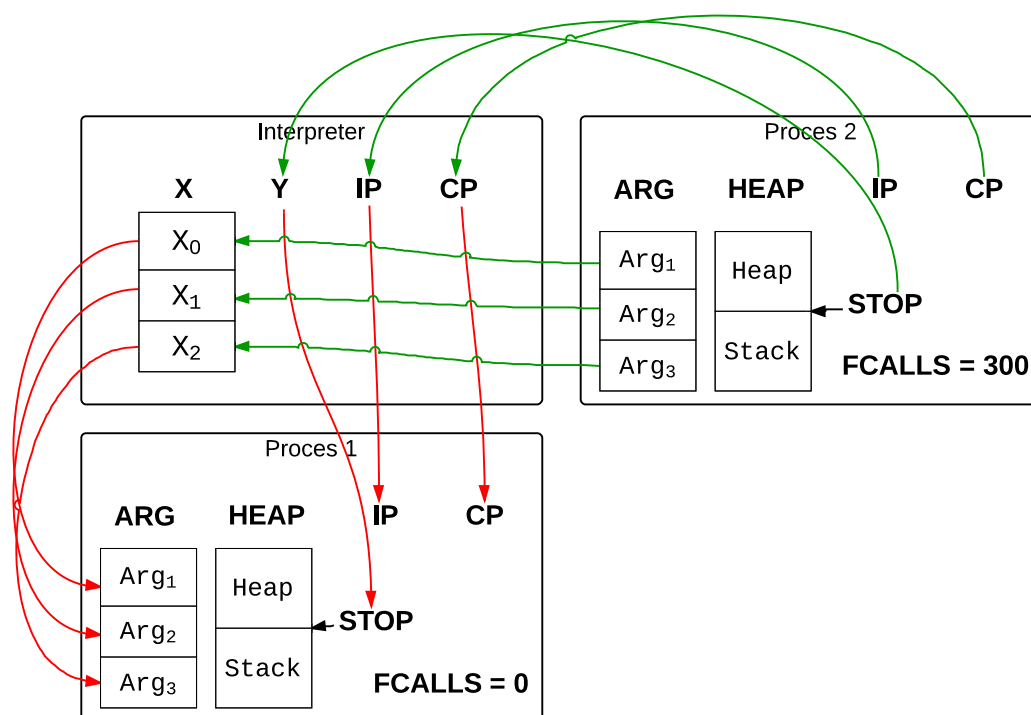
Pierwszym z nich jest spadek licznika redukcji w danym procesie do zera. Redukcja jest pojęciem wprowadzonym na potrzeby działania *schedulera* w maszynie wirtualnej BEAM. Jedna redukcja to wywołanie jednej funkcji (zewnętrznej lub wewnętrznej, operacja z rodziny **CALL\_\***). Domyślnie, w maszynie BEAM, gdy *scheduler* wznawia działanie procesu, ten dostaje możliwość wykonania 2000 redukcji zanim będzie musiał oddać dostęp do mocy obliczeniowej innemu procesowi. W maszynie zaimplementowanej w pracy, ze względu na mniejszą moc obliczeniową docelowej platformy, wartość ta została



zmniejszona do 300, podlega ona jednak konfiguracji. Aby nie doprowadzić do sytuacji, w której można zauważyć znaczącą rozbieżność w czasie dostępu do procesora pomiędzy różnymi procesami, redukcje liczone są także dla innych operacji. I tak np. funkcje wbudowane czy uruchomienie *garbage collector*a również modyfikują w pewien arbitralny sposób licznik redukcji w procesie.

Zmiana wykonywanego procesu może wystąpić tylko przed wywołaniem funkcji (czyli przed instrukcją z rodziny **CALL\_\***), tj. w chwili gdy mamy pewność że wszystkie lokalne zmienne zapisane są na stosie procesu a rejestry **X** wypełnione są argumentami funkcji.

Procedura zmiany aktualnie wykonywanego procesu polega na odpowiednim zapamiętaniu struktur interpretera w pamięci procesu przerywającego swoje działanie i odtworzenie tego samego rodzaju danych z pamięci procesu, który swoje działanie wznowia. Została ona zaprezentowana na rysunku 4.17. Wstrzymywane jest działanie procesu 1, w którego strukturze zapamiętywane są argumenty funkcji którą miał właśnie wywołać. Zapisywany jest także wskaźnik na szczyt jego stosu oraz wskaźniki instrukcji i adres powrotu. Na rysunku symbolicznie zostało to oznaczone czerwonymi strzałkami. Proces 2 z kolei wznowia swoje wykonanie. Licznik redukcji zostaje ustawiony ponownie na 300 i wykonywane są operacje odwrotne niż w przypadku procesu 1. Oznaczone one zostały symbolicznie strzałkami w kolorze zielonym.



Rysunek 4.17: Operacje wykonywane w trakcie zmiany wykonywanego procesu

Istnieje możliwość uruchomienia maszyny wirtualnej BEAM w architekturze SMP (*Symmetric Multiprocessing*), co domyślnie powoduje uruchomienie jednej instancji *schedulera* na jednym fizycznym wątku procesora. Synchronizację między planistami zapewniają w niej skomplikowane mechanizmy bę-

dące przedmiotem ciągłej optymalizacji. Maszynę zaimplementowaną w pracy można uruchomić jednak tylko z wykorzystaniem tylko jednego fizycznego wątku. Pomimo badań nad uruchomieniem *schedulera* mikrojądra FreeRTOS na architekturze wielordzeniowej [15], oficjalna dystrybucja wspiera obecnie tylko architekturę jednordzeniową.

Maszyna wirtualna BEAM posiada zaimplementowane priorytety procesów: **max**, **high**, **normal** oraz **low**. Procesy domyślnie uruchamiane są z priorytetem **normal**. Planista, w momencie gdy licznik redukcji aktualnie wykonywanego procesu osiągnie wartość 0 lub proces będzie oczekiwał na wiadomość w bloku `receive`, dokona wyboru kolejnego procesu do wykonania.

Proces może zostać wybrany spośród trzech dostępnych kolejek zawierających procesy o priorytecie:

- **max** — jeżeli w kolejce jest zakolejkowany przynajmniej jeden proces to jest będzie on wybranym jako kolejny do wykonania;
- **high** — jeżeli kolejka procesów o priorytecie **max** jest pusta a w tej kolejce znajduje się przynajmniej jeden proces to zostanie on wybrany do uruchomienia;
- **normal** i **low** — kolejka zawiera procesy o dwóch priorytetach, jednak procesy o priorytecie **low** zawierają specjalny licznik, który wskazuje na to ile razy proces musi zostać pominięty przy wyborze do uruchomienia zanim faktycznie zostanie uruchomiony. Wartość licznika dla tego rodzaju procesów po zakolejkowaniu wynosi 8. Procesy z tej kolejki mogą zostać wybrane do wznowienia przez *scheduler* tylko w sytuacji gdy kolejki z procesami o priorytetach **max** i **high** są puste.

Aktualny proces umieszczany jest na końcu kolejki odpowiadającej swojemu priorytetowi od razu (jeżeli zostaje zatrzymany z powodu wyczerpania się dostępnych redukcji) lub po otrzymaniu wiadomości lub przeterminowaniu (jeśli został zatrzymany na skutek wejścia w blok `receive`).

Wersja maszyny wirtualnej zaimplementowanej w pracy nie implementuje możliwości zmiany priorytetu uruchomionego procesu. Wszystkie procesy uruchamiane są z takim samym priorytetem. Jest to jednak funkcjonalność, której implementacja jest godna rozważenia w trakcie dalszego rozwoju projektu, biorąc pod uwagę fakt, że mikrojądro FreeRTOS pozwala na wybór priorytetu zadania a działanie planisty systemu jest bardzo podobne do ww. mechanizmu działania *schedulera* maszyny wirtualnej BEAM.

## 4.7. Funkcje wbudowane (*Built-In Functions*)

W niniejszym podrozdziale zaprezentowano zbiór zaimplementowanych funkcji wbudowanych w maszynę wirtualną dla systemu FreeRTOS. Z punktu widzenia programisty funkcja wbudowana jest zwykłą funkcją zaimplementowaną w zewnętrznym module, jej implementacja jest jednak częścią kodu maszyny wirtualnej w języku C. Nie jest zatem wykonywana ona jako kod pośredni przez interpreter, ale jako kod maszynowy danej architektury, co wpływa pozytywnie na szybkość jej wykonania. Każdej funkcji wbudowanej odpowiada wpis w tablicy funkcji eksportowanych, podobnie jak funkcjom eksportowanym z załadowanych modułów. Wpis ten zamiast wskaźnika do instrukcji kodu maszynowego do wykonania zawiera wskaźnik do funkcji zaimplementowanej w języku C.

### 4.7.1. Funkcje ogólne (moduł `erlang`)

Funkcje opisane w tej części podrozdziału zostały zaimplementowane w pliku źródłowym `erl_bif.c`.

Funkcja	Argumenty	Opis
<code>splus/2</code>	<code>Arg1 Arg2</code>	Zwraca <code>Arg1 + Arg2</code>
<code>sminus/2</code>	<code>Arg1 Arg2</code>	Zwraca <code>Arg1 - Arg2</code>
<code>stimes/2</code>	<code>Arg1 Arg2</code>	Zwraca <code>Arg1 * Arg2</code>
<code>spawn/3</code>	<code>M F A</code>	Startuje nowy proces, wykonujący funkcję <code>F</code> w module <code>M</code> z listą argumentów <code>A</code>
<code>spawn_link/3</code>	<code>M F A</code>	Startuje nowy proces, wykonujący funkcję <code>F</code> w module <code>M</code> z listą argumentów <code>A</code> . Łączy wystartowany proces z wywołującym <i>linkiem</i> .
<code>setelement/3</code>	<code>Index Tuple Element</code>	Zwraca nową krotkę, taką jak <code>Tuple</code> , ale z elementem <code>Element</code> na pozycji <code>Index</code>
<code>now/0</code>		Zwraca krotkę <code>{M, S, U}</code> zawierającą kolejną liczbę megasekund, sekund i mikrosekund jakie upłynęły od uruchomienia maszyny wirtualnej
<code>length/1</code>	<code>List</code>	Zwraca długość listy <code>List</code>
<code>plusplus/2</code>	<code>List1 List2</code>	Zwraca nową listę, stanowiącą złączenie list <code>List1</code> i <code>List2</code>
<code>div/2</code>	<code>Int1 Int2</code>	Zwraca wynik dzielenia całkowitoliczbowego argumentów
<code>rem/2</code>	<code>Int1 Int2</code>	Zwraca resztę z dzielenia argumentów
<code>element/2</code>	<code>N Tuple</code>	Zwraca <code>N</code> -ty element krotki <code>Tuple</code>
<code>exit/1</code>	<code>Reason</code>	Kończy działanie wywołującego procesu z powodem <code>Reason</code>
<code>process_flag/2</code>	<code>trap_exit true</code>	Wywołujący proces nie kończy działania gdy kończą działania procesy połączone, ale otrzymuje informacje o tym fakcie w postaci wiadomości
<code>self/0</code>		Zwraca <b>PID</b> wywołującego procesu
<code>send_after/3</code>	<code>Time Dest Mesg</code>	Ustawia przeterminowanie wysyłające wiadomość <code>Mesg</code> do procesu <code>Dest</code> za <code>Time</code> milisekund.

Tablica 4.3: Zaimplementowane funkcje wbudowane w module `erlang`

### 4.7.2. Listy (moduł `lists`)

Funkcje opisane w tej części podrozdziału zostały zaimplementowane w pliku źródłowym `erl_bif_lists.c`.

Funkcja	Argumenty	Opis
<code>reverse/1</code>	<code>List</code>	Zwraca nową listę będącą odwróconą listą <code>List</code>

Tablica 4.4: Zaimplementowane funkcje wbudowane w module `lists`

### 4.7.3. GPIO i obsługa przerwań (moduł `lpc_gpio`)

Funkcje opisane w tej części podrozdziału zostały zaimplementowane w pliku źródłowym `erl_bif_lpc.c`.

Funkcja	Argumenty	Opis
<code>output/2</code>	<code>Port Pin</code>	Ustawia pin <code>Pin</code> na porcie <code>Port</code> mikrokontrolera w tryb wyjścia
<code>high/2</code>	<code>Port Pin</code>	Ustawia stan wysoki na pinie <code>Pin</code> na porcie <code>Port</code>
<code>low/2</code>	<code>Port Pin</code>	Ustawia stan niski na pinie <code>Pin</code> na porcie <code>Port</code>

Tablica 4.5: Zaimplementowane funkcje wbudowane w module `lpc_gpio`

### 4.7.4. SPI (moduł `lpc_spi`)

### 4.7.5. Funkcje pomocnicze (moduł `lpc_debug`)

Funkcje opisane w tej części podrozdziału zostały zaimplementowane w pliku źródłowym `erl_bif_lpc.c`.

Funkcja	Argumenty	Opis
<code>dump_regs/0</code>		Drukuje zawartość rejestrów X
<code>dump_stack/0</code>		Drukuje zawartość stosu wywołującego procesu
<code>dump_heap/0</code>		Drukuje zawartość sterty wywołującego procesu
<code>print_info/0</code>		Drukuje informacje dotyczące wywołującego procesu
<code>print_term/1</code>	<code>Term</code>	Drukuje zawartość wyrażenia <code>Term</code>

Tablica 4.6: Zaimplementowane funkcje wbudowane w module `lpc_debug`

## 4.8. Garbage collector

Moduł opisany w niniejszym podrozdziale został zaimplementowany w pliku źródłowym `erl_gc.c`.

### 4.8.1. Pamięć zajmowana przez proces

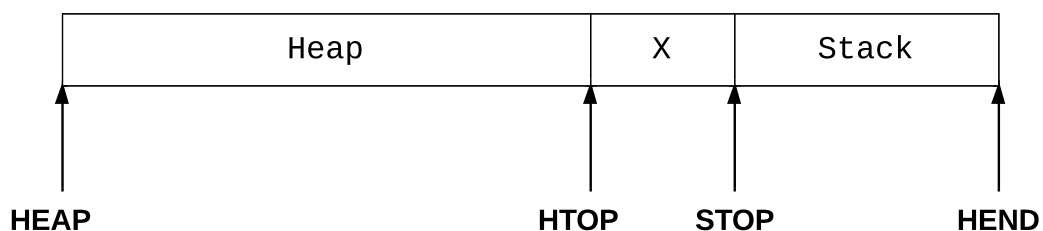
Blok pamięci zajmowany przez proces do przechowywania wyrażeń których używa nosi nazwę sterty i zarządzany jest automatycznie przez *garbage collector*. Obszar pamięci ten podzielony jest na dwie części:

- stertę, służącą do przechowywania danych złożonych (krotki, listy) lub opakowanych (**BOXED**), z których proces może korzystać nawet przez cały czas swojego życia;
- stos, którego zadaniem jest zachowywanie informacji pomiędzy wywołaniami funkcji, takich jak zmienne lokalne czy adresy powrotu.

Pojęcie sterty jest w niniejszej pracy używane wymiennie w odniesieniu albo do całego zaalokowanego dla procesu bloku pamięci albo do pierwszej wspomnianej jego części. Proces przechowuje kilka wskaźników dzięki którym możliwa jest identyfikacja odpowiednich fragmentów pamięci i wykonywanie operacji na nich. Należą do nich:

- **HEAP**, oznaczający początek całego bloku pamięci, a zarazem i sterty;
- **HTOP**, oznaczający koniec sterty;
- **STOP**, oznaczający początek stosu;
- **HEND**, oznaczający koniec bloku pamięci, a zarazem i stosu.

Graficzne odniesienie wskaźników do miejsc w pamięci procesu zostało zaprezentowane na rysunku 4.18. Jak można zauważyć, wyrażenia dodawane do sterty procesu umieszczane są począwszy od początku całego bloku pamięci. Z kolei wyrażenia dodawane na stos umieszczane są od końca bloku, w kierunku jego początku. Wskazywanie przez wskaźniki **HTOP** oraz **STOP** na to samo miejsce w pamięci oznacza, że na sterce procesu skończyło się miejsce i konieczne jest uruchomienie *garbage collector*.



Rysunek 4.18: Struktura pamięci zajmowanej przez proces wraz ze wskaźnikami na poszczególne struktury

Sztywność zaalokowana dla procesu może mieć jeden z pewnych ustalonych z góry rozmiarów, należący do ciągu:

$$h_n = \begin{cases} 12 & \text{dla } n = 1, \\ 38 & \text{dla } n = 2, \\ h_{n-1} + h_{n-2} + 1 & \text{dla } n > 2 \end{cases} \quad (4.1)$$

Rozmiar ten wyrażony jest w słowach maszynowych i dla maszyny BEAM domyślnie wynosi 233 (6. wyraz ciągu). W maszynie zaimplementowanej w pracy domyślnym, początkowym rozmiarem sztywności procesu jest 12 słów maszynowych ze względu na mniejsze zapotrzebowanie na pamięć, wynikające z przeznaczenia aplikacji.

#### 4.8.2. Algorytm Cheney'a

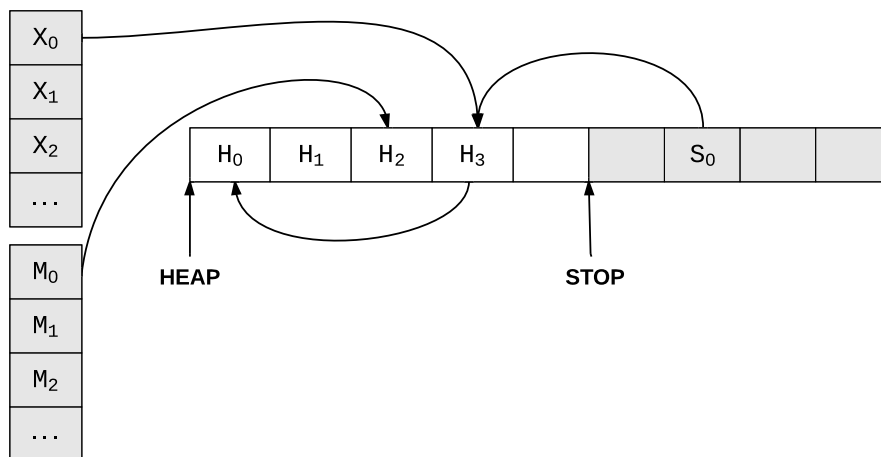
Algorytmem odświeżania wykorzystywanym w maszynie wirtualnej Erlanga jest algorytm Cheney'a [8]. Jest on algorytmem kopiującym, którego działanie polega na zaalokowaniu nowego bloku pamięci i przeniesieniu do niego wszystkich obecnie używanych obiektów. Kolejnym krokiem jest przepisanie wszystkich istniejących referencji do poprzedniego bloku pamięci. Na końcu następuje zwolnienie starego bloku pamięci. Ten prosty algorytm pasuje do filozofii języka, gdzie każdy uruchomiony proces korzysta tylko i wyłącznie z pamięci zajmowanej przez samego siebie. Dzięki temu możliwe jest uruchamianie *garbage collector'a* w kontekście każdego procesu z osobna. Nie powoduje to zatrzymań działania całego uruchomionego systemu na czas odświeżania sztywności procesu. Algorytm zostanie uruchomiony w sytuacji, gdy na sterce nie ma wolnej liczby słów maszynowych potrzebnej do zaalokowania w danej chwili lub zostanie on wywołany *explicite* przez funkcję `erlang:garbage_collect/0-2`.

Działanie algorytmu Cheney'a uruchomionego w procesie erlangowego zostało zaprezentowano na rysunkach od 4.19 do 4.25. Do zilustrowania sposobu działania algorytmu posłużono się abstrakcją trójkolorową, służącą do znalezienia obiektów których pamięć można zwolnić, wykorzystującą do ich oznaczania trzech kolorów:

- białego, oznaczającego że stan obiektu jest w tym momencie nieznanym, ale nie napotkano jeszcze żadnego obiektu odwołującego się do niego;
- szarego, obiekt jest wykorzystywany i nie można go usunąć, ale nie przetworzono jeszcze obiektów zależnych od niego;
- czarnego, obiekt, podobnie jak obiekty od niego zależne, są wykorzystywane.

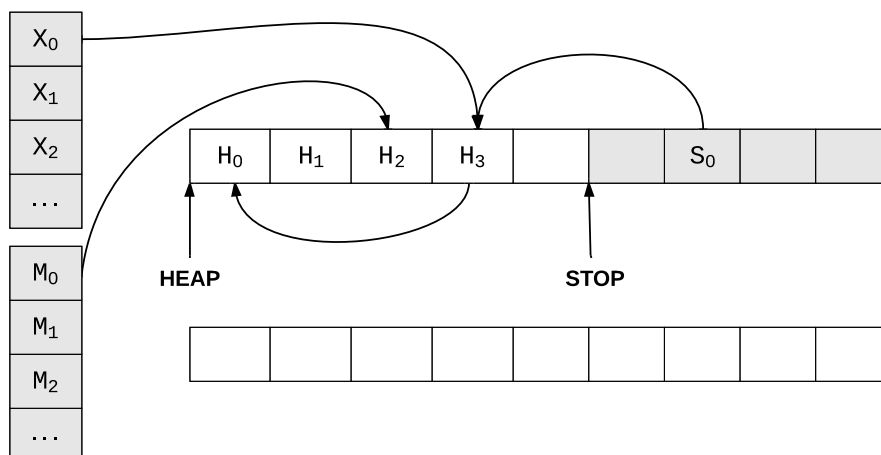
Celem algorytmu odświeżania jest osiągnięcie takiego stanu, w którym wszystkie obiekty będą oznaczone kolorem czarnym (obiekty wykorzystywane) lub białym (obiekty niewykorzystywane które można usunąć z pamięci). Na początku działania algorytmu, początkowy zbiór wyrażeń oznaczony jest kolorem szarym. Ze względu na kopiujący charakter algorytmu, kolor obiektów na dotychczasowej sterce procesu będzie zawsze biały. Obiekty przekopiowane na nową sztywność będą z kolei tylko szare lub czarne.

Do początkowego zbioru wyrażeń, które odnosić będą się do używanej pamięci, w zaimplementowanej maszynie należą: stos procesu, używane rejestry  $X$  i kolejka wiadomości (rys. 4.19). W maszynie wirtualnej BEAM do tego zbioru należy jeszcze kilka innych struktur, jak np. słownik procesu.



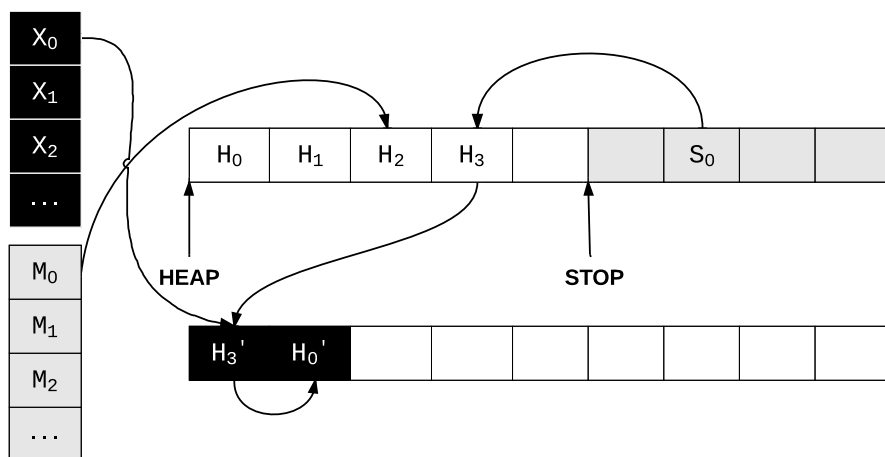
Rysunek 4.19: Sterta poddawana odświeżeniu. Szary zbiór stanowi stos procesu, rejestry  $X$  i kolejka wiadomości procesu.

Pierwszym krokiem działania algorytmu jest zaalokowanie sterty procesu o takiej samej długości jak dotychczasowa (rys. 4.20). Następnie przetwarzane są kolejne elementy zbioru szarego, a wyrażenia do których zawierają referencję kopiowane są na nową stertę. Tak jest w przypadku wyrażenia  $H_3$ , do którego odnosi się rejestr  $X_0$ . Wyrażenie to zostało przekopiowane na nową stertę i referencja do nowego wyrażenia została zapisana w tym rejestrze. Obiekt  $H_3$  miał w tym momencie kolor szary. Dlatego, że on sam zawierał referencję do wyrażenia  $H_0$ , to wyrażenie także musiało zostać przeniesione na nową stertę. Przeniesiony obiekt nie zawierał już referencji do żadnego innego obiektu, dlatego zaraz po oznaczeniu go kolorem szarym można było zmienić jego kolor na czarny.



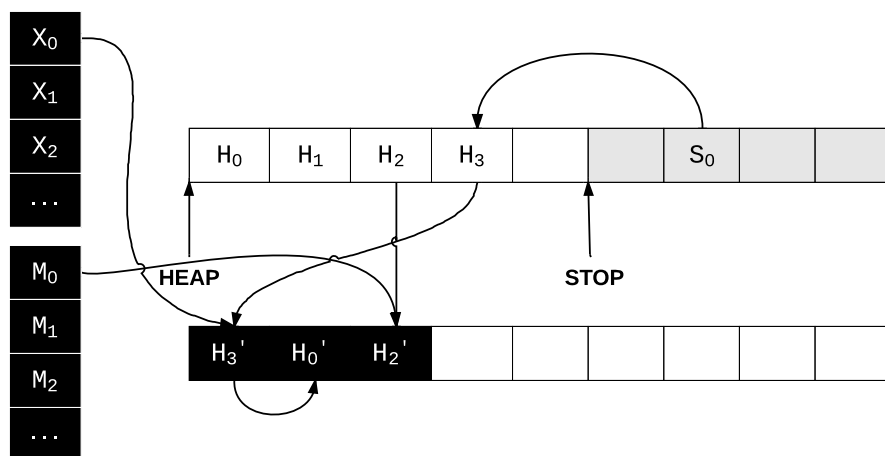
Rysunek 4.20: Alokacja nowej sterty o takim samym rozmiarze jak dotychczasowa.

Dopiero po podmianie referencji w wyrażeniu  $H_3$ ' do obiektu  $H_0$ ' można było oznaczyć ten pierwszy kolorem czarnym (rys. 4.21). Należy również zauważyć, że staremu obiektowi  $H_3$  przypisano jego kopię na nowej sterce. Zabieg ten wykonany został po to, aby kolejne obiekty ze zbioru szarego odnoszące się do niego nie kopiowały go na nową stertę, ale zaktualizowały referencję do jego istniejącej już kopii.



Rysunek 4.21: Przeniesienie wyrażeń mających źródło w rejestrach na nową stertę.

Analogicznie, na nową stertę przeniesione zostało wyrażenie  $H_2$ , do którego odnosiła się wiadomość  $M_0$  (rys. 4.22).

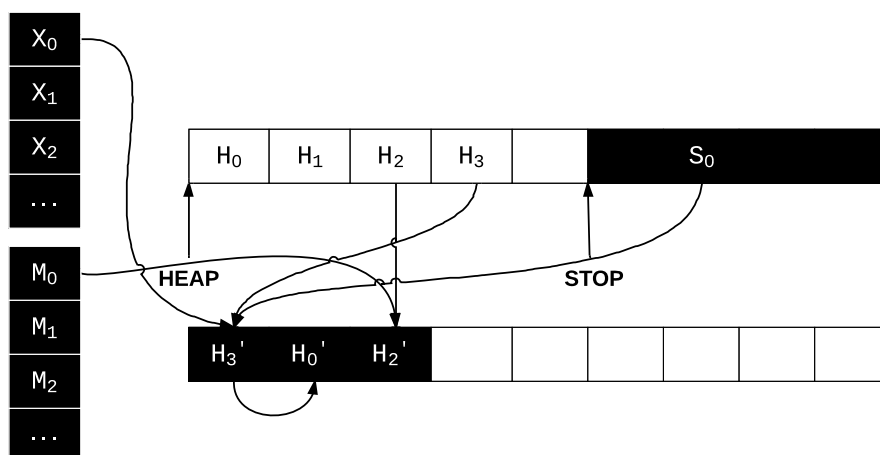


Rysunek 4.22: Przeniesienie wyrażeń mających źródło w kolejce wiadomości.

Ostatnim analizowanym zbiorem szarym był stos procesu, w którym wyrażenie  $S_0$  odnosiło się do, przeniesionego już wcześniej, wyrażenia  $H_0$ . Dlatego też w tym kroku wystarczające było przepisanie referencji do kopii tego obiektu do wyrażenia na stosie (rys. 4.23).

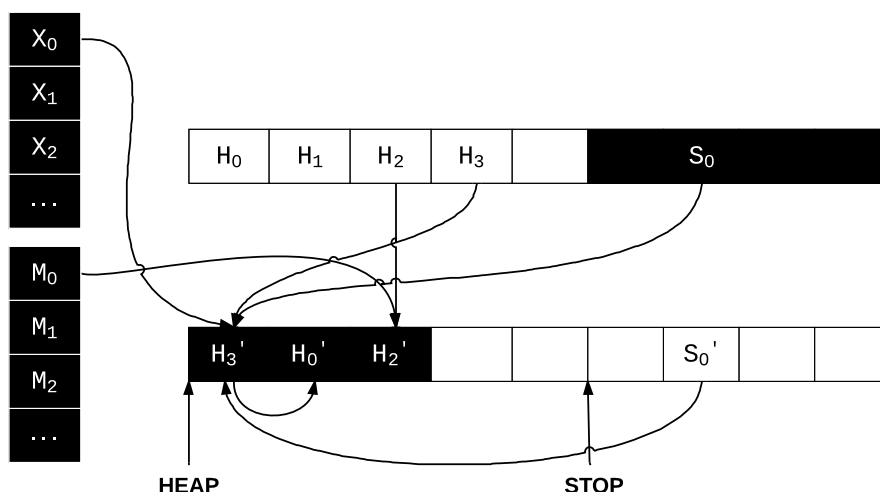
Przedostatnim krokiem działania algorytmu było skopiowanie stosu procesu, gdyż znajduje się on zawsze w tym samym bloku pamięci co sterta, a więc również zostaje usunięty w trakcie działania





Rysunek 4.23: Podmiana wskaźnika do już przeniesionego wyrażenia.

*garbage collector*. Dokonano także przepisania odpowiednich wskaźników oznaczających początek sterty i stosu procesu (rys. 4.24).

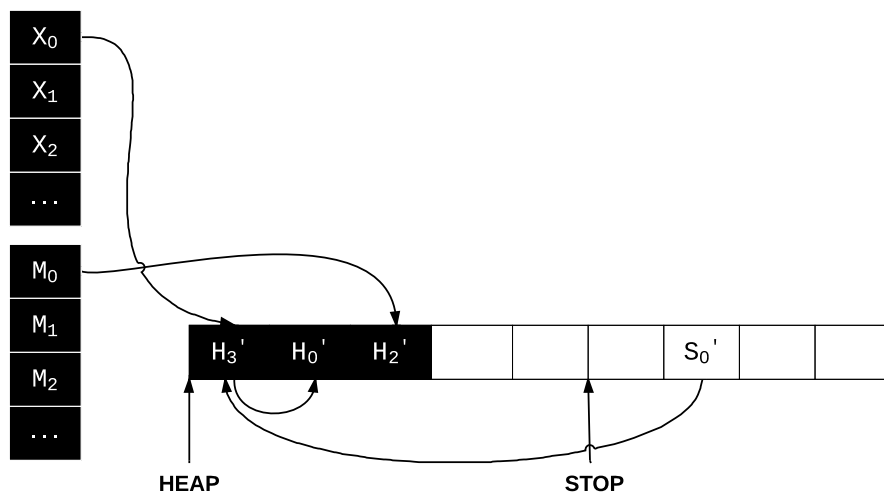


Rysunek 4.24: Kopiowanie stosu na nową stertę i przepisanie odpowiednich wskaźników.

Końcowym krokiem działania algorytmu było zwolnienie dotychczas używanego bloku pamięci. Jak można zauważyć, na nowej stercie nie ma wyrażenia  $H_1$ , które nie było już wykorzystywane w momencie uruchomienia algorytmu (rys. 4.25).

Jeżeli dojdzie do sytuacji, w której po uruchomieniu *garbage collector* na stercie nie będzie odpowiedniej ilości miejsca dla nowych obiektów, dokonane zostanie rozszerzenie sterty. Polega ono na zaalokowaniu nowego obszaru pamięci, o kolejnym możliwym rozmiarze niż obecny i przeniesieniu stosu oraz wszystkich wyrażeń ze sterty. W takiej sytuacji konieczne jest również przepisanie referencji obiektów do nowych miejsc w pamięci. Po zakończeniu tego procesu źródłowa sterta jest zwalniana.

Analogiczną operację wykonuje się w sytuacji, gdy pamięć zajmowana przez nowe obiekty zajmuje co najwyżej pewien ustalony rozmiar. W maszynie wirtualnej został on ustalony na połowę rozmiaru



Rysunek 4.25: Zwolnienie pamięci zajmowanej przez starą stertę.

starej sterty. W takiej sytuacji sterta procesu jest zmniejszana do najbliższego wymaganemu rozmiarowi i, podobnie jak w przypadku rozszerzania sterty, zawartość sterty i stos przenoszone są do nowego bloku pamięci.

### 4.8.3. Podejście generacyjne w maszynie BEAM

Maszyna BEAM w procesie odświeżania korzysta dodatkowo z podejścia generacyjnego. Opiera się ono na obserwacji, że większość utworzonych obiektów usuwana jest z pamięci stosunkowo wcześnie, a obiekty które „przetrwaly” przynajmniej jedno uruchomienie *garbage collector*a prawdopodobnie przetrwają i kolejne. Prowadzi to do podziału obiektów na dwie generacje, którym przypisane są osobne sterty: „młoda” i „stara”. Pierwsza z nich zawiera niedawno utworzone obiekty, które nie zostały poddane jeszcze żadnemu procesowi odświeżania. Druga z kolei przechowuje obiekty, które zostały utworzone w pamięci przed przynajmniej jednym uruchomieniem *garbage collector*a. Odświeżanie „starej” sterty odbywa się z założenia zdecydowanie rzadziej niż jest to w przypadku obszaru pamięci z nowymi obiektami. Częstość ta w maszynie wirtualnej BEAM jest parametrem konfigurowalnym przez zmienną środowiskową `ERL_FULLSWEEP_AFTER`, której wartość mówi co ile uruchomień *garbage collector*a na „młodej” generacji obiektów uruchomiony on zostanie również dla „starej” generacji. Im większa wartość, tym sam proces odświeżania trwa zdecydowanie krócej, proces wykorzystuje jednak więcej pamięci.

W maszynie wirtualnej zaimplementowanej w pracy nie została zaimplementowana ta optymalizacja algorytmu odświeżania.

## 4.9. Mechanizmy zarządzania czasem

Moduł opisany w niniejszym podrozdziale został zaimplementowany w pliku źródłowym `erl_time.c`.

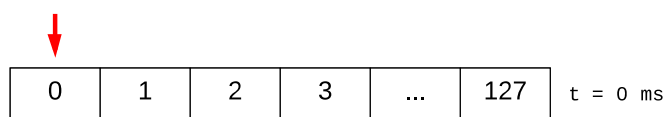
Język Erlang zapewnia dwie konstrukcje pozwalające na ustawienie przeterminowania po upływie zadanego czasu, po którym przez maszynę wirtualną zostanie wykonana pewna czynność:

- konstrukcja `receive ... after ... end` — zawieszająca wykonywanie się procesu aż do momentu otrzymania wiadomości, kiedy to przeterminowanie zostaje usunięte z listy aktywnych lub do upłynięcia ustalonego czasu w milisekundach, kiedy proces wznowia działanie wykonując najpierw kod w bloku `after`;
- wywołanie funkcji wbudowanej `erlang:send_after/3` — kolejkującej wysłanie wiadomości do wskazanego procesu po upłynięciu zadanego czasu. Usunięcia tak utworzonego przeterminowania można dokonać poprzez wywołanie innej funkcji wbudowanej `erlang:cancel_timer/1`.

Wymienione funkcjonalności związane z zarządzaniem przeterminowaniami zostały zaimplementowane w maszynie wirtualnej opisywanej w pracy.

Strukturą danych służącą do zapamiętania aktywnych przeterminowań jest koło czasowe, będące tablicą dwukierunkowych list przeterminowań, o określonej z góry długości. Sposób działania mechanizmu został zaprezentowany na rysunkach od 4.26 do 4.28.

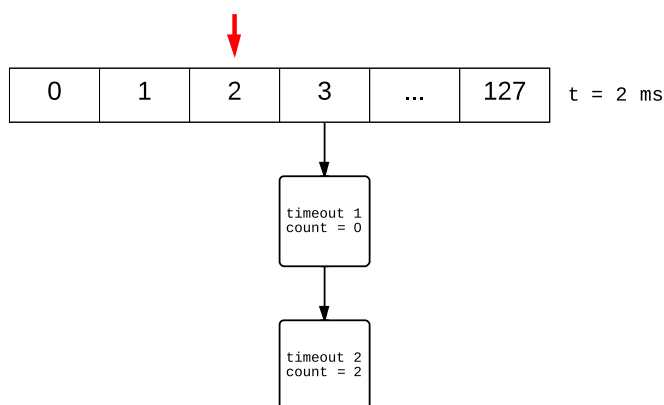
Na rysunku 4.26 zaprezentowano wygląd struktury w momencie uruchomienia maszyny wirtualnej. Koło czasowe ma tutaj 128 pól, a rozdzielczość każdego z nich to 1 milisekunda. Wskaźnik, oznaczony na rysunkach czerwoną strzałką, zapamiętuje aktualne pole koła czasowego. W momencie aktualizacji czasu wskaźnik ten jest przesuwany o odpowiednią liczbę pól, a w przypadku dojścia do końca tablicy ustawiany jest on ponownie na jej początku.



Rysunek 4.26: Stan koła czasowego w momencie uruchomienia systemu.

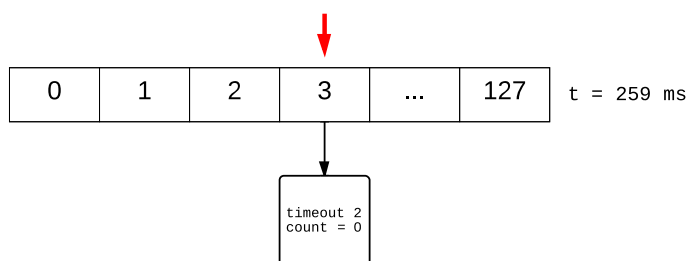
Na rys. 4.27 zaprezentowano wygląd koła czasowego po upływie 2 milisekund od uruchomienia systemu. W międzyczasie do koła czasowego dodano dwa przeterminowania:

- do zrealizowania za 1 milisekundę — ponieważ aktualne pole w kole czasowym to 2, przeterminowanie zostało dodane na pozycji 3 i zostanie zrealizowane przy kolejnej aktualizacji czasu;
- do zrealizowania za 257 milisekund — realizacja przeterminowania może nastąpić dopiero po dwukrotnym przejściu wskaźnika przez całe koło czasowe (256 ms) i dojścia do pozycji nr 3 w tablicy, dlatego w przypadku tego przeterminowania wartość licznika ustawiona została na 2.



Rysunek 4.27: Koło czasowe z dodanymi dwoma przeterminowaniami.

Rysunek 4.28 przedstawia stan koła czasowego w 259. milisekundzie. Wskaźnik, oznaczony czerwoną strzałką, w porównaniu do stanu zaprezentowanego na poprzednim rysunku wykonał dwa pełne przejścia przez całą tablicę koła czasowego. Każda aktualizacja czasu (przesunięcie wskaźnika) pociąga za sobą aktualizację przeterminowań na pozycjach mijanych przez wskaźnik. Jeżeli licznik przeterminowania jest równy zeru oznacza to, że przeterminowanie powinno zostać zrealizowane i usunięte z listy na danej pozycji. W przeciwnym wypadku, przeterminowanie przeznaczone jest do zrealizowania w którymś z kolejnych przejść wskaźnika przez daną pozycję koła czasowego. Dlatego też licznik danego przeterminowania zostaje zmniejszony o 1.



Rysunek 4.28: Koło czasowe z jednym przeterminowaniem, które zostanie zrealizowane przy kolejnej aktualizacji czasu.

Jedyną różnicą w budowie koła czasowa pomiędzy maszyną zaimplementowaną w pracy a maszyną BEAM jest jej rozmiar. W maszynie BEAM składa się ona aż z 65536 pozycji. Rozmiar ten, ze względu na ograniczony rozmiar dostępnej pamięci, w niniejszej maszynie został ograniczony do minimum.

Aspektem różniącym obie maszyny w działaniu mechanizmów zarządzania czasem jest również sposób aktualizacji czasu, a co za tym idzie częstość aktualizacji wskaźnika koła czasowego. W maszynie wirtualnej BEAM wskaźnik aktualizowany jest na podstawie zegara systemowego pomiędzy wykonywaniem kolejnych procesów z kolejek, a także gdy nie ma w nich żadnych procesów do uruchomienia. Z kolei w zaimplementowanej maszynie wirtualnej rola aktualizacji czasu przejęta została przez fizyczny zegar aktualizowany taktowaniem o częstotliwości 1 kHz. Zegar uruchamia przerwanie w mikrokontrolerze.

lerze, a kod je obsługujący aktualizuje pozycję wskaźnika koła czasowego i aktualizuje jego stan wraz z realizacją odpowiednich przeterminowań. W idealnym przypadku więc pozycja wskaźnika będzie aktualizowana z częstotliwością równą rozdzielczości koła czasowego (co 1 milisekundę), jednak opóźnienia w obsłudze przerwania mogą prowadzić do rzadszej jego aktualizacji.

W niniejszej maszynie wirtualnej została zaimplementowana została także funkcja wbudowana `erlang:now/0`, różniąca się jednak działaniem od jej odpowiednika w maszynie BEAM tym, że zwraca czas jaki upłynął od momentu uruchomienia systemu, nie zaś od początku 1 stycznia 1970 r. Czas pozyskiwany jest z, innego niż w przypadku aktualizacji czasu w kole czasowym, fizycznego zegara mikrokontrolera LPC1769, aktualizowanego taktowaniem o częstotliwości 1 MHz.

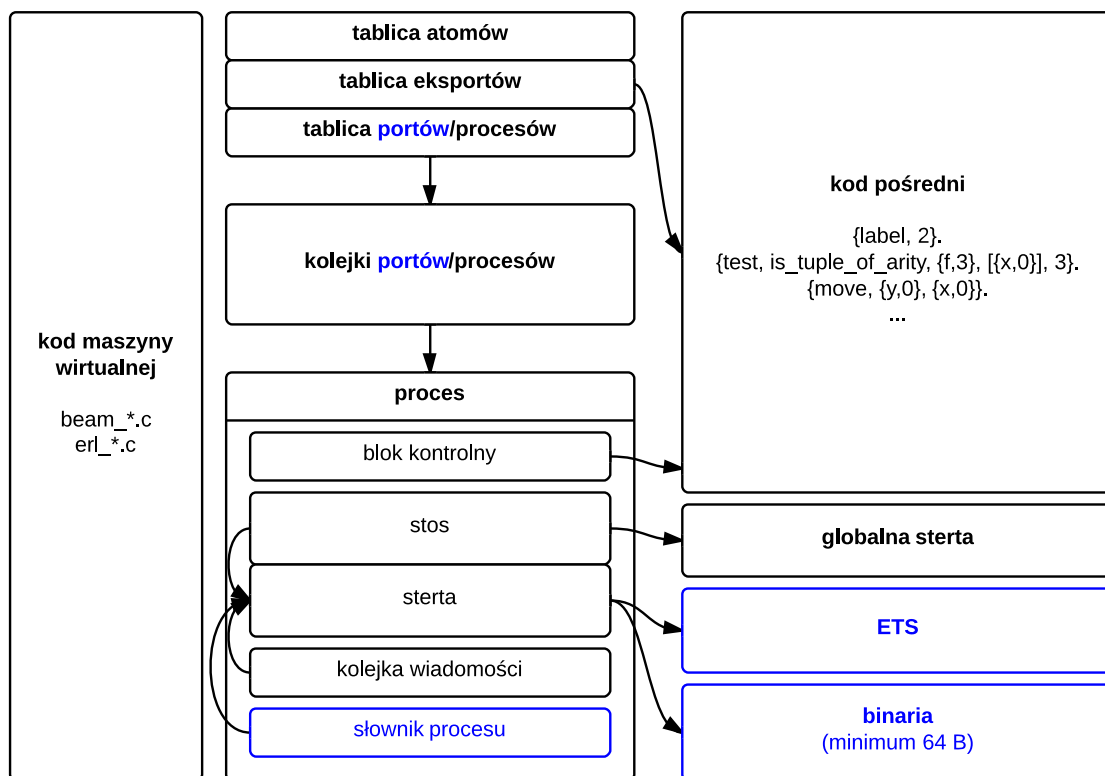
## 4.10. Podsumowanie różnic z maszyną BEAM

Na elementy, z których składa się maszyna wirtualna Erlanga można popatrzeć z punktu widzenia poszczególnych obszarów pamięci przez nią zajmowanych.

Do elementów tych należą:

- skompilowany kod wykonywalny modułów opisanych w niniejszym rozdziale;
- tablice: atomów, eksportowanych funkcji, portów i procesów;
- kolejki procesów i portów, na podstawie których *scheduler* decyduje o wyborze kolejnego procesu do wykonania;
- uruchomione procesy z informacjami kontrolnymi, kolejkami wiadomości i pamięcią do nich należącą;
- załadowany kod pośredni modułów, który wykonywany jest w kontekście procesów;
- globalna sarta, na której umieszczane są stałe wyrażenia pochodzące z plikami z modułów;
- tablice ETS (*Erlang Term Storage*) stanowiące w maszynie wirtualnej mutowalny obszar pamięci, w postaci bazy danych klucz-wartość;
- sarta, na której przechowywane są dane typu binarnego, których długość wynosi przynajmniej 64 bajtów. Dane te mogą być współdzielone przez procesy, których liczba przechowywana jest przez licznik referencji, na którego podstawie *garbage collector* zwalnia nieużywane już obszary pamięci.

Na rysunku 4.29 zaprezentowany został powyższy podział, z uwzględnieniem elementów które posiada zarówno maszyna BEAM jak i maszyna zaimplementowana w pracy, które zostały oznaczone kolorem czarnym. Kolorem niebieskim z kolei oznaczone zostały elementy, które nie zostały zaimplementowane w maszynie na system FreeRTOS. Strzałki na diagramie przedstawiają fakt przechowywania wskaźników na elementy w jednym obszarze pamięci przed drugi.



Rysunek 4.29: Elementy maszyny wirtualnej Erlanga jako obszary pamięci.

Elementami, które nie zostały zaimplementowane w pracy są: tablice ETS oraz słownik procesu, których użycie nie jest konieczne do implementacji w pełni funkcjonalnych modułów w języku Erlang. Nie pozostały również zaimplementowane sterta binariów oraz tablice i kolejki portów, ze względu na niezaimplementowanie tych typów danych w maszynie.

## 5. 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 B. Kody źródłowe aplikacji zostały umieszczone na płycie CD dołączonej do pracy.

### 5.1. Silnia

#### 5.1.1. 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. 89).

Listingi 5.1 oraz 5.2 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 5.1: 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 5.2: 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).

### 5.1.2. 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 5.1.

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).

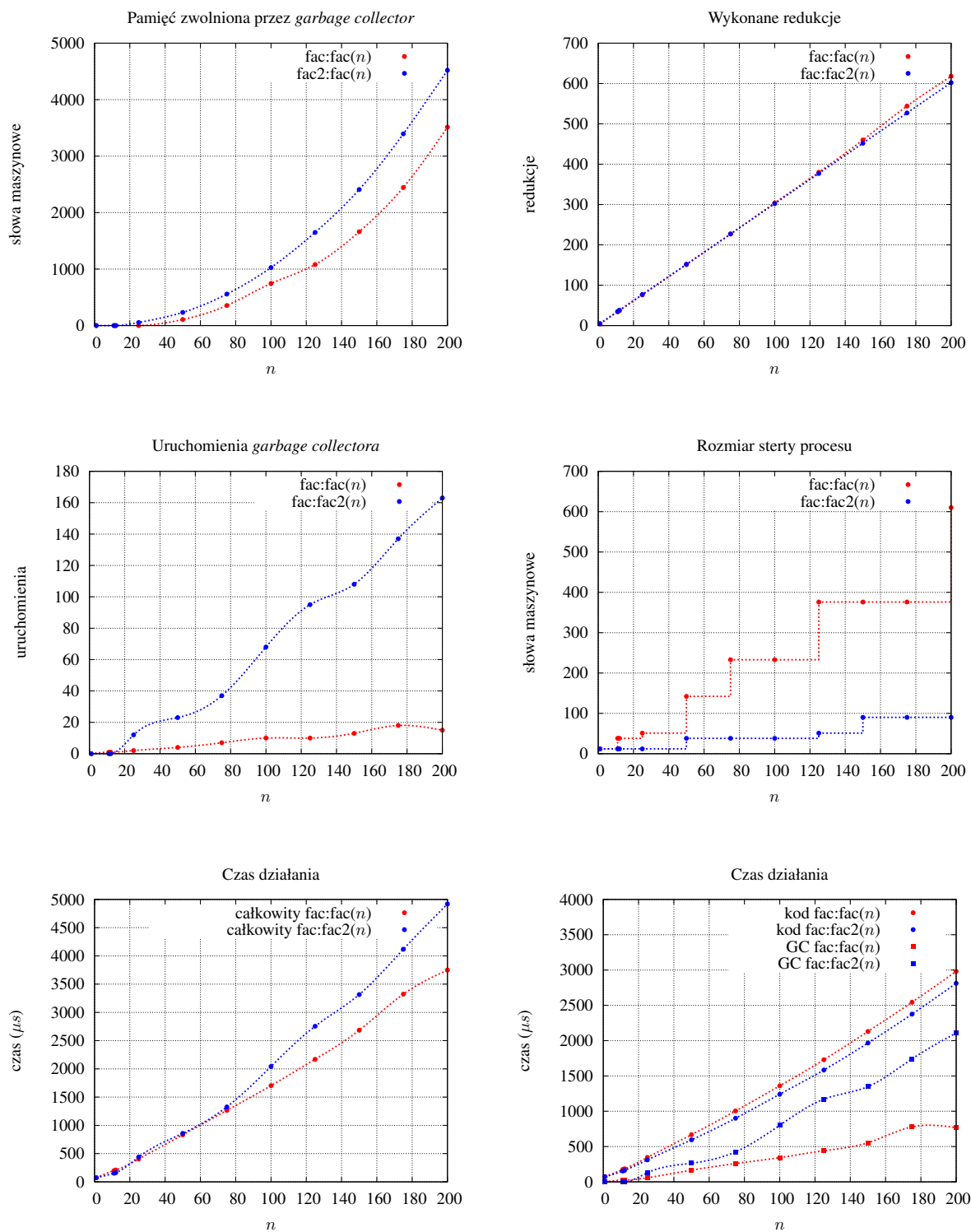
Dla porównania, obie powyższe funkcje obliczające silnię z 200 na maszynie wirtualnej BEAM na komputerze z systemem operacyjnym MacOS X i procesorem Intel Core i7, wykonały się w ok. 1800  $\mu$ s. Należy wspomnieć, że początkowy rozmiar sterty procesu został ustawiony na 12 słów maszynowych, tak jak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

### 5.1.3. 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 5.1: Porównanie wyników uruchomienia modułów `fac` oraz `fac2` na implementowanej maszynie wirtualnej.

## 5.2. Kontrola diod LED przez procesy

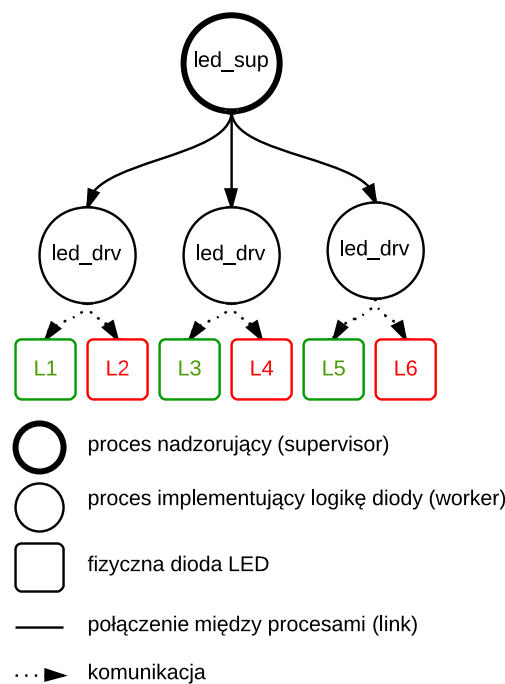
### 5.2.1. Cel aplikacji

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

W tym celu zaimplementowano dwa moduły: `led_sup` i `led_drv`, których zadaniem była kontrola diod LED, przez zaimplementowane funkcje kontrolujące GPIO, w dwóch kolorach: czerwonym i zielonym. W systemie uruchomiony był jeden proces implementujący logikę pierwszego z modułów oraz osiem procesów - 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 sekundowym, wysyłał do trzech losowych procesów wiadomość o zmianie stanu diody. Dodatkowo, w logice procesu ustawione zostało przeterminowanie, efektem którego było zakończenie się działania procesu w sytuacji, gdy żadna wiadomość kontrolująca stan diody nie przyszła do skrzynki odbiorczej procesu w odstępie 2 sekund.

Stan systemu wizualizowało dodatkowo osiem czerwonych diod, które były zgaszone w momencie gdy procesy kontrolujące odpowiadający 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 dwóch sekund po otrzymaniu wiadomości o zakończeniu się jego poprzednika.

Zależności między procesami uruchomionymi w ramach przykładowej aplikacji zostały zaprezentowane na rysunku 5.2. Fizyczna realizacja połączeń diod, sterowanych stanem niskim, do mikrokontrolera wykorzystanego w przykładzie przedstawiona została na rysunku 5.3. Na obu rysunkach diody zostały oznaczone tymi samymi symbolami.



Rysunek 5.2: Struktura procesów w aplikacji i kontrolowanych przez nie diod

### 5.2.2. Uzyskane wyniki

### 5.2.3. Wnioski

## 5.3. Sterownik RFM73

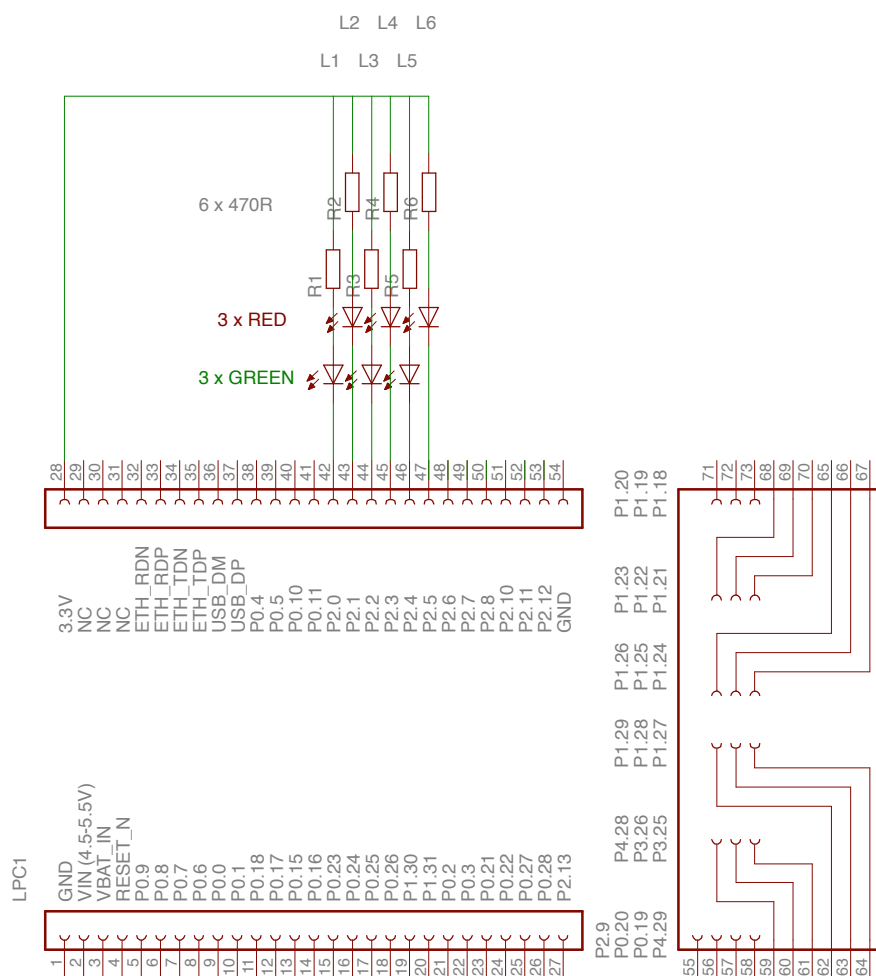
### 5.3.1. Cel aplikacji

Celem aplikacji było zaimplementowanie biblioteki sterownika do modułu radiowego RFM73 [9], 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.

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;



Rysunek 5.3: Schemat podłączenia diod LED do płytki prototypowej z mikrokontrolerem LPC1769

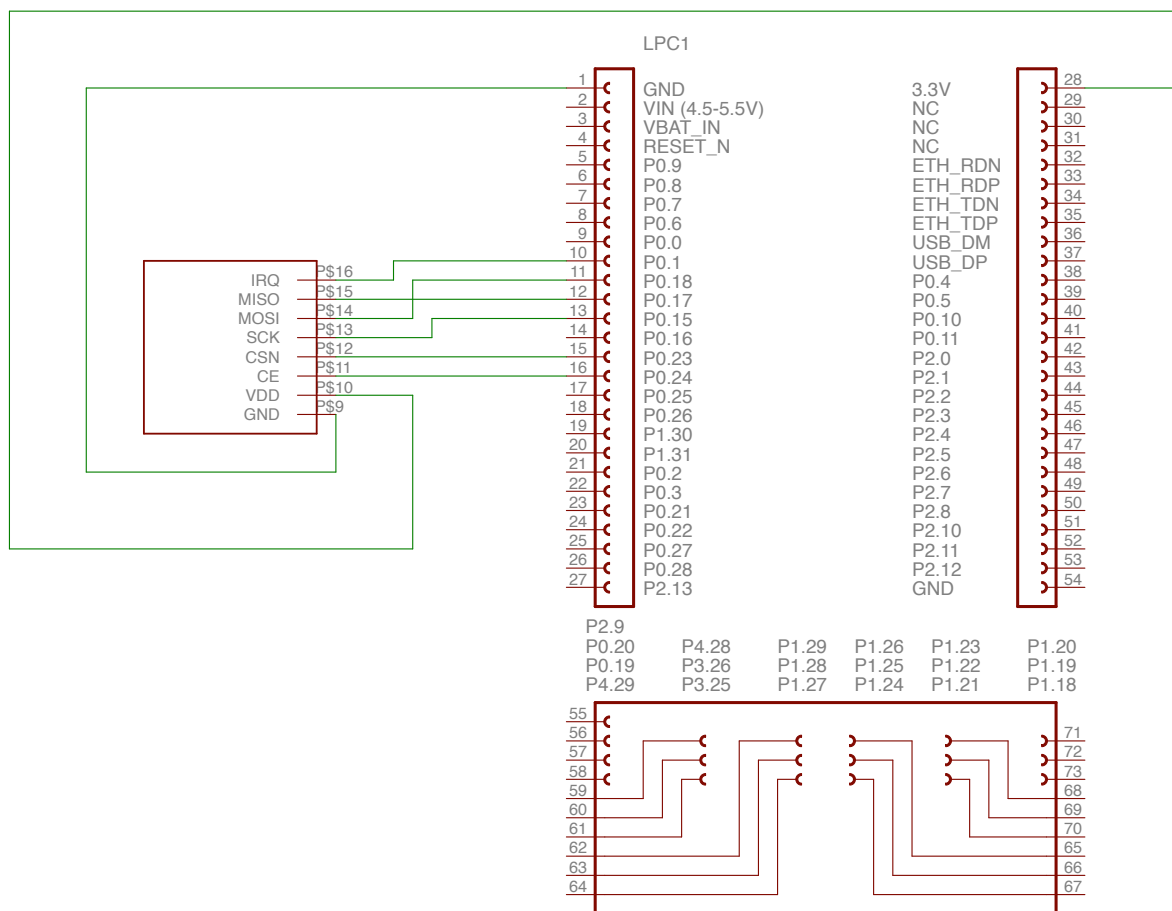
– urządzenia peryferyjne do komputerów osobistych.

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 5.4.

### 5.3.2. 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 urzą-



Rysunek 5.4: Schemat podłączenia modułu RFM73 do płytki prototypowej z mikrokontrolerem LPC1769

dzenia 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.

### 5.3.3. Wnioski

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

## 6. Podsumowanie

### 6.1. Wnioski

### 6.2. Dalszy rozwój projektu

W ramach przyszłych prac warto dokonać próby rozwoju projektu w następujących aspektach:

- rozbudowa narzędzia budującego aplikacje, tak aby możliwe było zbudowanie maszyny wirtualnej ze skompilowanymi modułami wraz z odpowiednim portem systemu FreeRTOS w jednym kroku;
- implementacja pozostałych opkodów;
- implementacja pozostałych typów danych, takich jak binaria czy referencje;
- implementacja własnej logiki planisty, co pozwoli na oszczędność pamięci używanej przez zadanie systemu FreeRTOS;
- implementacja protokołu *Distributed Erlang* [2] i integracja go ze stosem TCP/IP dedykowanym dla systemu FreeRTOS;
- umożliwienie ładowania kodu do pamięci za pomocą *Distributed Erlang*, dzięki czemu wkompilowywanie kodu modułów w kod maszyny wirtualnej przestanie być konieczne.





## A. Zawartość dołączonej płyty CD

Na dołączonej do niniejszej pracy płycie CD znajdują się następujące pliki:

- plik `praca.pdf` zawierający treść niniejszej pracy w formacie Portable Document Format;
- katalog `tex` zawierający źródło niniejszej pracy w  $\text{\LaTeX}$ ;
- katalog `src` zawierający kod źródłowy implementacji maszyny wirtualnej opisanej w pracy;
- katalog `prezentacja` zawierający źródła i plik wyjściowy w formacie PDF prezentacji do niniejszej pracy;
- katalog `examples` zawierający przykładowe aplikacje opisane w rozdziale 5;
- katalog `builder` zawierający narzędzie do budowy aplikacji do uruchomienia na maszynie wirtualnej, opisane w dodatku B;
- katalog `FreeRTOS_Library` zawierający kod źródłowy mikrojądra FreeRTOS w wersji 8.0.1;
- katalog `CMSISv1p30_LPC17xx` zawierający zestaw nagłówków pomocniczych dla mikrokontrolera LPC1769.



## B. Kompilator aplikacji

Dodatek zawiera opis działania pomocnicznego narzędzia utworzonego dla potrzeb pracy, służącego do przygotowywania skompilowanych modułów Erlanga do włączenia w maszynę wirtualną uruchamianą pod systemem FreeRTOS.

### B.1. Opis aplikacji

Maszyna wirtualna Erlanga dla systemu FreeRTOS implementowana w ramach pracy, na obecnym poziomie zaawansowania wymaga wkompilowania bajtkodu modułu w maszynę wirtualną. Kod źródłowy maszyny oczekuje specjalnie przygotowanego pliku `modules.h`, który zawiera kod modułów a także moduł, funkcję i argumenty, które zaczną być wykonywane w momencie startu maszyny. Cały kod źródłowy maszyny wirtualnej może zostać następnie skompilowany wraz z mikrojądrem FreeRTOS, za pomocą platformy NXP LPCXpresso [19], przeznaczonej do rozwijania i programowania aplikacji na mikrokontrolery serii LPC.

Niniejsze narzędzie służy właśnie do generowania pliku `modules.h`, którego skopiowanie do katalogu z kodem źródłowym maszyny wirtualnej spowoduje włączenie wybranych modułów do maszyny wirtualnej. Oprócz tego, przed wygenerowaniem właściwego pliku wyjściowego, narzędzie dokonuje rozpakowania fragmentów kodu modułu, które zostały spakowane algorytmem **zlib**. Funkcjonalność ta została uwzględniona ze względu na fakt, że moduł ładujący kod w maszynie opisywanej w pracy nie posiada zaimplementowanego algorytmu dekompresji tego formatu.

### B.2. Kompilacja narzędzia

Narzędzie zostało zaimplementowane w języku Erlang, dlatego też do jego kompilacji wymagany jest kompilator Erlanga. Narzędziem budującym projekt jest `rebar`, który również jest wymagany.

Aby skompilować aplikację, wystarczy użyć komendy `make` w głównym katalogu aplikacji. Plikiem wyjściowym będzie plik wykonywalny `freertos`, stanowiący samodzielną aplikację. Możliwe zatem jest skopiowanie go do wygodnego w użyciu folderu, np. uwzględnionego w domyślnej ścieżce dla aplikacji.

### B.3. Użycie narzędzia

Aplikacja jest aplikacją konsolową, która jako obowiązkowe argumenty przyjmuje listę plików źródełowych Erlanga, które zostaną skompilowane do pliku nagłówkowego maszyny wirtualnej.

Aby skompilować aplikację opisaną w rozdziale 5.1, należy po wejściu do katalogu z przykładem użyć komendy `freertos *.erl` (zakładając że skompilowane wcześniej narzędzie znajduje się w ścieżce domyślnej). Efektem polecenia będzie wygenerowany plik `modules.h`, który następnie należy skompilować razem z maszyną wirtualną.

Istnieje możliwość wyboru pliku wyjściowego poprzez użycie opcji `-o` lub `--output`, jeżeli požądane jest wygenerowanie innego pliku niż domyślnego `modules.h` w bieżącym katalogu.

Istnieje również możliwość wyboru początkowej funkcji programu (domyślnie jest to `main:main()`) z użyciem opcji `-m` lub `--module` dla modułu, `-f` lub `--function` dla funkcji oraz `-a` lub `--argument` dla argumentów.

Opis użycia narzędzia można zawsze wyświetlić uruchamiając aplikację `freertos` bez użycia żadnych argumentów.

## **C. Konfiguracja parametrów maszyny wirtualnej**



## D. Kompilacja kodu źródłowego

Dodatek opisuje kolejne kroki, z jakich składa się proces otrzymywania skompilowanego kodu pośredniego maszyny wirtualnej BEAM z kodu źródłowego napisanego w języku Erlang. Oprócz tego dodatek dokumentuje, na potrzeby projektu, zawartość pliku ze skompilowanym kodem pośrednim. Format pliku nie jest objęty oficjalną dokumentacją języka ze względu na dużą zmienność pomiędzy kolejnymi wersjami kompilatora i maszyny wirtualnej.

### D.1. Wprowadzenie

Narzędzia przeznaczone do generacji wszystkich form pośrednich kodu źródłowego opisanych w niniejszym rozdziale zostały napisane w języku Erlang. Dostępne są one w pakiecie aplikacji `compiler` dostarczanej wraz z maszyną wirtualną BEAM.

### D.2. Kod źródłowy

Listingi D.1 oraz D.2 prezentują prosty, przykładowy moduł zaimplementowany w języku Erlang, którego zadaniem jest obliczanie silni za pomocą funkcji ogonowo-rekurencyjnej.

Elementy takie jak używanie rekordu do przechowywania akumulatora funkcji czy zwracanie krotki informującej o błędzie zaciemniają nieco sposób działania funkcji. W tym przykładzie zostały one jednak zawarte w celu przeanalizowania przekształceń, jakim poddawany jest kod źródłowy w trakcie poszczególnych kroków kompilacji.

```
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 D.1: Plik fac.erl

```

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

```

Listing D.2: Plik fac.hrl

### D.3. Preprocessing

Pierwszym krokiem w procesie otrzymywania kodu maszynowego Erlanga jest wstępne przetwarzanie wykonywane przez preprocesor kompilatora.

Krok ten jest dwuetapowy. W pierwszym, z plikiem modułu łączone są pliki, które zostały do niego włączone poprzez użycie dyrektywy `include`. W miejscach dołączania zewnętrznych plików dodawane są także dyrektywy `file` po to, aby przy debugowaniu skompilowanego modułu można było zidentyfikować z którego pliku pochodzą dane fragmenty kodu. Na tym etapie podstawiane są również wartości makr. Jeżeli w opcjach kompilacji zdefiniowana jest operacja `{parse_transform, Module}` modyfikująca drzewo składniowe modułu to zostanie ona również wykonywana w tym kroku, przekazując do dalszej kompilacji wynik działania funkcji `Module:parse_transform/2`.

Kod modułu po tym kroku został przedstawiony na listingu D.3. Wygenerowanie kodu w tej postaci jest możliwe przy użyciu kompilatora z opcją `-P:erlc -P fac.erl`.

Kolejnym krokiem jest rozwinięcie rekordów do krotek (rekordy są tylko lukrem składniowym języka Erlang, upraszczającym operacje na krotkach o dużej liczbie pól) oraz dołączenie funkcji `module_info/0` oraz `module_info/1`, które znajdują się w każdym skompilowanym module. Funkcje te zwracają informacje o skompilowanym module, takie jak np. eksportowane funkcje.

Kod modułu po wykonaniu tej operacji został przedstawiony na listingu D.4. Wygenerowanie kodu w tej postaci jest możliwe przy użyciu kompilatora z opcją `-E:erlc -E fac.erl`.

```

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 D.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 D.4: Moduł fac po drugim przetworzeniu

## D.4. Drzewo składniowe

Formatem, jakiego używa kompilator Erlanga do wykonywania poszczególnych kroków kompilacji jest drzewo składniowe modułu. Warto w tym miejscu przypomnieć, że programista rów-

niez może ingerować w drzewo składniowe modułu, używając wspomnianej już opcji kompilacji `parse_transform`.

Drzewo składniowe dla przykładowego modułu `fac`, po przetworzeniu wstępnym, zostało przedstawione na listingu D.5.

```

1  [{attribute,1,file,{"fac.erl",1}},
2      {attribute,1,file,{"fac.hrl",1}},
3      {attribute,7,file,{"fac.erl",7}},
4      {function,8,fac,1,
5          [{clause,8,
6              [{tuple,8,[{atom,8,factorial},{integer,8,0},{var,8,'Acc'}]}],
7              [],
8              [{var,9,'Acc'}]},
9          {clause,10,
10             [{tuple,10,
11                 [{atom,10,factorial},{var,10,'N'},{var,10,'Acc'}]}],
12             [],
13             [{call,11,
14                 {atom,11,fac},
15                 [{tuple,11,
16                     [{atom,11,factorial},
17                     {op,11,'-',{var,11,'N'},{integer,11,1}},
18                     {op,11,'*',{var,11,'N'},{var,11,'Acc'}}]}]}],
19             {clause,12,
20                 [{var,12,'N'}],
21                 [{call,12,
22                     {remote,12,{atom,12,erlang},{atom,12,is_integer}},
23                     [{var,12,'N'}]}]},
24                 [{call,13,
25                     {atom,13,fac},
26                     [{tuple,13,
27                         [{atom,13,factorial},
28                         {var,13,'N'},
29                         {integer,1,1}}]}]}]},
30             {clause,14,
31                 [{var,14,'N'}],
32                 [{call,14,
33                     {remote,14,{atom,14,erlang},{atom,14,is_binary}},
34                     [{var,14,'N'}]}]},
35                 [{call,15,
36                     {atom,15,fac},
37                     [{call,15,
38                         {remote,15,
39                             {atom,15,erlang},
40                             {atom,15,binary_to_integer}},
41                         [{var,15,'N'}]}]}]}],
42             {clause,16,
```

```

43         [{var, 16, '_' }],
44         [],
45         [{tuple, 17,
46           [{atom, 17, error}, {string, 17, "Invalid argument"}]}]}],
47     {function, 0, module_info, 0,
48       [{clause, 0, [], []},
49        [{call, 0,
50          {remote, 0, {atom, 0, erlang}, {atom, 0, get_module_info}},
51          [{atom, 0, fac}]}]}]}],
52     {function, 0, module_info, 1,
53       [{clause, 0,
54         [{var, 0, 'X' }],
55         [],
56         [{call, 0,
57           {remote, 0, {atom, 0, erlang}, {atom, 0, get_module_info}},
58           [{atom, 0, fac}, {var, 0, 'X' }]}]}]}]}]

```

Listing D.5: Drzewo składniowe modułu fac

Drzewo składniowe modułów może zostać także wykorzystane w sytuacji, jeżeli chcemy utworzyć kompilator innego języka programowania, który kompilowałby kod źródłowy w tym języku do kodu maszynowego Erlanga. W efekcie możliwe będzie uruchamianie programów napisanych w tym języku na dowolnej maszynie wirtualnej Erlanga. Wydaje się to być dobrym pomysłem dla języków, które mogą wynieść dużo korzyści z uruchamiania programów w nich napisanych na maszynie mającej takie właściwości jak BEAM. Przykładem tego może być język Elixir [17].

## D.5. Core Erlang

Kolejnym krokiem kompilacji jest transformacja kodu do innego języka - Core Erlang. Jest to język funkcyjny, składnią przypominający język Erlang. Jednak ze względu na uproszczenie składni, pozwala on na łatwiejszą maszynową optymalizację i konwersję do kodu pośredniego maszyny wirtualnej (bajtkodu).

Kod rozważanego w tym dodatku modułu, w języku Core Erlang, został umieszczony na listingu D.6.

```

1 module 'fac' ['fac' /1,
2             'module_info' /0,
3             'module_info' /1]
4 attributes []
5 'fac' /1 =
6   %% Line 4
7   fun (_cor0) ->
8     case _cor0 of
9       <1> when 'true' ->

```

```

10      %% Line 5
11      1
12      %% Line 6
13      <N>
14      when call 'erlang':'is_integer'
15          (_cor0) ->
16      let <_cor1> =
17          %% Line 7
18          call 'erlang':'-'
19          (N, 1)
20      in let <_cor2> =
21          %% Line 7
22          apply 'fac'/1
23          (_cor1)
24      in %% Line 7
25          call 'erlang':'*'
26          (N, _cor2)
27      %% Line 8
28      <_X_Other> when 'true' ->
29          %% Line 9
30          'not_integer'
31      end
32      'module_info'/0 =
33      fun () ->
34          call 'erlang':'get_module_info'
35          ('fac')
36      'module_info'/1 =
37      fun (_cor0) ->
38          call 'erlang':'get_module_info'
39          ('fac', _cor0)
40      end

```

Listing D.6: Moduł fac w Core Erlang

## D.6. Kod pośredni - bajtkod

Dopiero z modułu w postaci Core Erlang generowany jest bajtkod - kod maszynowy rozumiany przez maszynę wirtualną Erlanga. Język maszynowy zawiera instrukcje z określonego zestawu instrukcji, których pełna lista wraz z opisem argumentów znajduje się w dodatku E.

Wygenerowanie kodu w tej postaci jest możliwe przy użyciu kompilatora z opcją `-S`: `erlc -S fac.erl`.

Wygenerowany kod pośredni dla przykładowego modułu silni, w formie listy krotek, został zawarty na listingu D.7. Na listingu zawarty jest kod pośredni dla każdej z funkcji modułu (oznaczonej krotką

{function, ...}). Pierwszym elementem każdej krotki wewnątrz funkcji jest nazwa operacji, a kolejnymi argumenty tej operacji.

```

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 D.7: Bytecode modułu fac

Kod modułu w postaci kodu pośredniego jest jeszcze na zbyt wysokim poziomie abstrakcji, aby mógł bezpośrednio zostać zrozumiany przez maszynę wirtualną. Wszystkie argumenty operacji użyte w kodzie, takie jak np. odnośniki do etykiet czy atomy muszą zostać zapisane w odpowiednich tablicach i ich indeksy w odpowiedniej formie mogą dopiero zostać użyte jako właściwe argumenty operacji. Oprócz tego same nazwy operacji muszą zostać zamienione na odpowiadające im opkody. Czynności te wykonywane są w kolejnym kroku - generacji pliku binarnego z kodem modułu, opisanym w kolejnej sekcji.

## D.7. Plik binarny BEAM

Efektem przetworzenia kodu pośredniego, wyrażonego w postaci krotek, jest plik binarny w formacie IFF [16], w formacie zrozumiałym przez maszynę wirtualną BEAM. Maszyna ta wykorzystuje tego rodzaju pliki do ładowania kodu poszczególnych 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 D.1 zaprezentowana została ogólna struktura pliku binarnego ze skompilowanym modułem.

	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 D.1: Struktura pliku modułu BEAM

Każdy plik binarny BEAM powinien zawierać przynajmniej 4 następujące fragmenty (*chunki*). Obok opisu każdego fragmentu, w nawiasie podano ciąg znaków będący jego identyfikatorem w binarnym pliku modułu:

- tablica atomów wykorzystywanych przez moduł (*Atom*);
- kod pośredni danego modułu (*Code*);
- tablica zewnętrznych funkcji używanych przez moduł (*ImpT*);

- tablica funkcji eksportowanych przez moduł (`ExpT`).

Ponadto, w pliku mogą znajdować się następujące fragmenty:

- tablica funkcji lokalnych dla danego modułu (`LocT`);
- tablica lambd wykorzystywanych przed modułem (`FunT`);
- tablica stałych wykorzystywanych przed modułem (`LitT`);
- lista atrybutów modułu (`Attr`);
- lista dodatkowych informacji o kompilacji modułu (`CInf`);
- tablica linii kodu źródłowego modułu (`Line`);
- drzewo syntaktyczne modułu (`Abst`).

W przypadku każdego rodzaju fragmentu, obszar pamięci 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 poszczególne fragmenty powinny występować w pliku binarnym.

### D.7.1. Tablica atomów

Tablica atomów zawiera listę wszystkich atomów, które używane są przez dany moduł. W trakcie ładowania kodu modułu przez maszynę wirtualną, atomy, które nie występowały we wcześniej załadowanych modułach, zostają wstawione do globalnej tablicy atomów (w postaci tablicy z hashowaniem).

Ponieważ długość atomu zapisana jest na jednym bajcie, nazwa atomu może mieć maksymalnie 255 znaków.

Fragment pliku binarnego z tablicą atomów reprezentowany jest przez napis `Atom`. Struktura danych fragmentu zaprezentowana jest w tabeli D.2.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Ilość atomów w tablicy atomów															
4	32	Dł. atomu 1								Nazwa atomu 1 w ASCII							
...	...	Dł. atomu 2								Nazwa atomu 2 w ASCII							
...	...	...															

Tablica D.2: Struktura tablicy atomów w pliku BEAM



### D.7.2. Kod pośredni

Sekcja z kodem pośrednim zawiera faktyczny kod wykonywalny modułu, który jest interpretowany przez maszynę wirtualną w trakcie uruchomienia systemu. Szczegółowy opis reprezentacji i znaczenia opkodów i ich argumentów zawarty został w dodatku E.

Fragment pliku z kodem identyfikowany jest przez napis `Code`. Struktura danych fragmentu zawarta została w tabeli D.3.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0x000010															
4	32	Numer wersji formatu kod (w Erlangu R16 - 0x00000000)															
8	64	Maksymalny numer operacji (do sprawdzenia kompatybilności)															
12	96	Liczba etykiet w kodzie modułu															
16	128	Liczba funkcji eksportowanych z modułu															
20	160	Opkod 1								Argument 1							
...	...	...								Argument N							
...	...	Opkod 2								Argument 1							
...	...	...															

Tablica D.3: Struktura kodu pośredniego w pliku BEAM

### D.7.3. Tablica importowanych funkcji

Fragment pliku binarnego z tablicą importowanych funkcji zawiera informacje o funkcjach zaimplementowanych w innych modułach, które są wykorzystywane przez moduł.

Identyfikowany jest on przez napis `ImpT`. Struktura danych fragmentu zawarta została w tabeli D.4.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Liczba importowanych funkcji															

4	32	Indeks atomu z nazwą modułu 1
8	64	Indeks atomu z nazwą funkcji 1
12	96	Arność funkcji 1
16	128	Indeks atomu z nazwą modułu 2
...	...	...

Tablica D.4: Struktura tablicy importowanych funkcji w pliku BEAM

#### D.7.4. Tablica eksportowanych funkcji

Fragment pliku binarnego z tablicą eksportowanych funkcji zawiera informacje o funkcjach z modułu, które widoczne są z poziomu innych modułów.

Identyfikowany jest on przez napis `ExpT`. Struktura danych fragmentu zawarta została w tabeli D.5.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Liczba eksportowanych funkcji															
4	32	Indeks atomu z nazwą funkcji 1															
8	64	Arność funkcji 1															
12	96	Etykieta początku kodu funkcji 1															
16	128	Indeks atomu z nazwą funkcji 2															
...	...	...															

Tablica D.5: Struktura tablicy eksportowanych funkcji w pliku BEAM

#### D.7.5. Tablica funkcji lokalnych

Fragment pliku binarnego z tablicą lokalnych funkcji zawiera informacje o funkcjach zaimplementowanych w module (w tym `lambda`), które wykorzystywane są tylko przez ten moduł i nie są widoczne z poziomu innych modułów.

Identyfikowany jest on przez napis `LOC`T. Struktura danych fragmentu zawarta została w tabeli D.6.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Liczba lokalnych funkcji															
4	32	Indeks atomu z nazwą funkcji 1															
8	64	Arność funkcji 1															
12	96	Etykieta początku kodu funkcji 1															
16	128	Indeks atomu z nazwą funkcji 2															
...	...	...															

Tablica D.6: Struktura tablicy lokalnych funkcji w pliku BEAM

#### D.7.6. Tablica `lambd`

Fragment pliku binarnego z tablicą `lambd` zawiera informacje o obiektach funkcyjnych, które wykorzystywane są przez ten moduł.

Lambdy identyfikowane są poprzez atomy, które powstały przez złączenie nazwy funkcji, w której zostały zdefiniowane oraz kolejny indeks lambdy zdefiniowanej w danej funkcji. Np. kolejne obiekty funkcyjne zdefiniowane w funkcji `foo/1` będą identyfikowane przez atomy `-foo/1-fun-0-`, `-foo/1-fun-1-` itd.

Fragment pliku tablicą lambdy identyfikowany jest przez napis `Fun`T. Struktura danych fragmentu zawarta została w tabeli D.7.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Liczba <code>lambd</code> w module															
4	32	Indeks atomu z identyfikatorem lambdy 1															
8	64	Arność lambdy 1															
12	96	Etykieta początku kodu lambdy 1															

16	128	Indeks lambdy 1 (0x00)
20	160	Liczba wolnych zmiennych w lambdzie 1
24	192	Wartość skrótu z drzewa syntaktycznego kodu lambdy 1
28	224	Indeks atomu z identyfikatorem lambdy 2
...	...	...

Tablica D.7: Struktura tablicy lambd w pliku BEAM

### D.7.7. Tablica stałych

Fragment pliku binarnego z tablicą lambd stałych zawiera informacje o stałych (listy, napisy, duże liczby) które wykorzystywane są przez ten moduł.

Właściwa lista wartości stałych (od bajtu 4 do końca fragmentu) przechowywana jest w pliku w postaci skompresowanej algorytmem **zlib**. Podany rozmiar w bajtach dotyczy nieskompresowanej tablicy stałych. Stałe zapisane są w formacie binarnym w formacie *External Term Format*, opisanym w dokumencie [3].

Fragment pliku z tablicą identyfikowany jest przez napis `LitT`. Struktura danych fragmentu (w zdekompresowanej postaci) zawarta została w tabeli D.8.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Rozmiar tablicy w bajtach															
4	32	Liczba stałych															
8	64	Rozmiar stałej 1 w bajtach															
12	96	Stała 1 w External Term Format															
...	...	Rozmiar stałej 2 w bajtach															

...	...	...
-----	-----	-----

Tablica D.8: Struktura tablicy stałych w pliku BEAM

### D.7.8. Lista atrybutów modułu

Fragment pliku binarnego z listą atrybutów modułu zawiera listę dwójek (tzw. *proplistę*) ze wszystkimi dodatkowymi atrybutami, z jakimi został skompilowany dany moduł (np. informacje o wersji czy autorze). Lista ta zapisana jest binarnie w postaci *External Term Format*.

Fragment ten reprezentowany jest przez napis `Attr`.

### D.7.9. Lista dodatkowych informacji o kompilacji modułu

Fragment pliku binarnego z listą informacji o kompilacji modułu zawiera *proplistę* z informacjami dotyczącymi kompilacji, takimi jak: ścieżka pliku z kodem źródłowym, czas kompilacji, wersja kompilatora czy użyte opcje kompilacji. Informacje te zapisane są binarnie w postaci *External Term Format*.

Fragment ten reprezentowany jest przez napis `CInf`.

### D.7.10. Tablica linii kodu źródłowego modułu

Fragment pliku binarnego z informacjami o liniach kodu źródłowego modułu zawiera informacje dla instrukcji `line/1` maszyny wirtualnej o pliku źródłowym i linii, z której pochodzi aktualnie wykonywany fragment kodu. Informacje te wykorzystywane są przy generowaniu stosu wywołań przy wystąpieniu błędu lub wyjątku. Funkcjonalność ta została wprowadzona dopiero w wersji R15 maszyny wirtualnej BEAM.

Jeżeli kompilowany plik jest na etapie preprocessingu łączony z innymi plikami z kodem źródłowym (poprzez użycie atrybutu `include`) to informacja o tych plikach zostanie zawarta w tym fragmencie. Domyślnie, kompilowany plik nie zostanie uwzględniony i zostanie przydzielony mu indeks 0.

Numer linii koduje się przy użyciu tagu `0001`, jak w przypadku argumentów instrukcji maszyny wirtualnej, opisanych w sekcji E.1. Rozróżnienie pliku, z którego pochodzi linia odbywa się za pomocą zapamiętania, z którego pliku pochodziła ostatnia linia. Domyślnie jest to plik o indeksie 0. Jeżeli dochodzi do zmiany aktualnego pliku, kolejny numer linii poprzedzony jest indeksem pliku z którego pochodzi, zakodowanym przy użyciu tagu `0010` (jak w sekcji E.1). Dlatego też numer linii może zawierać w pliku binarnym 1 lub 2 bajty.

Fragment pliku z tablicą identyfikowany jest przez napis `Line`. Struktura danych fragmentu zawarta została w tabeli D.9.

	Oktet	0	1
--	-------	---	---

Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Wersja (0x0000000)															
4	32	Flagi (0x0000000)															
8	64	Liczba instrukcji line w kodzie modułu															
12	96	Liczba linii z kodem w plikach modułu															
16	128	Liczba plików z kodem modułu															
20	160	Numer linii (1 lub 2 B)								Numer linii (1 lub 2 B)							
...	...	...															
...	...	Długość nazwy pliku 1								Nazwa pliku 1 w ASCII							
...	...	...															
...	...	Długość nazwy pliku 2								Nazwa pliku 2 w ASCII							
...	...	...															

Tablica D.9: Struktura tablicy linii kodu źródłowego w pliku BEAM

### D.7.11. Drzewo syntaktyczne modułu

Plik z modułem zawiera fragment pliku źródłowego z drzewem syntaktycznym pliku z kodem źródłowym o ile został skompilowany z opcją `debug_info`. Fragment ten identyfikowany jest przez napis `Abst.`

Zawartością fragmentu jest drzewo syntaktyczne modułu, w postaci opisanej w sekcji D.4 zakodowane w formacie *External Term Format*.

## E. Lista instrukcji maszyny wirtualnej BEAM

Dodatek zawiera listę instrukcji maszyny wirtualnej BEAM, jakie może zawierać skompilowany kod pośredni przez nią wykonywany oraz sposób zapisu argumentów dla instrukcji.

Kod danej operacji zajmuje zawsze 1 bajt w pliku ze skompilowanym kodem pośrednim modułu. Argumenty mogą zajmować więcej przestrzeni, zgodnie z opisem w sekcji E.1.

Kolejność bajtów w zapisie kodu pośredniego to zawsze *big endian*.

### E.1. Typy argumentów

Argumentem jest zawsze liczba całkowita, reprezentująca wartość liczbową bądź indeks w odpowiedniej tablicy z wartościami (pierwszym indeksem takiej tablicy jest 0). W związku z tym argumenty mogą być różnego typu. Aby rozróżnić argument jednego typu od drugiego poddaje się je odpowiedniemu tagowaniu. Operację wykonuje się bezpośrednio na argumentie, jeśli jest dostatecznie mały, lub na odpowiednim nagłówku poprzedzającym argument. Rozróżnienie to jest spowodowane oszczędnością rozmiaru kodu pośredniego, który musi być przechowywany w pamięci.

Każdy z tagów, które zostały wymienione w tabeli E.1, jest możliwy do zapisania przy użyciu 3 bitów, które zajmują najmniej znaczące bity argumentu. Jednak w kodowaniu binarnym do zapisu typu używane są dodatkowo 1 lub 2 bity. Dzięki nim możliwe jest rozróżnienie pomiędzy argumentami zapisanymi przy użyciu różnej liczby bajtów.

Tagowanie odbywa się za pomocą następującej operacji:

$$(0000XXXX \ll N)_{(2)} \vee 000SSTTT_{(2)},$$

gdzie  $XXXX_{(2)}$  jest tagowaną liczbą,  $N = 4$  lub  $5$ ,  $SS_{(2)}$  są dodatkowymi bitami znakującymi rozmiar argumentu, a  $TTT_{(2)}$  jest danym tagiem.

Tag		Typ
binarnie	dziesiętnie	
000	0	uniwersalny indeks, np. do tablicy stałych
001	1	liczba całkowita
010	2	indeks do tablicy atomów

011	3	numer rejestru X maszyny wirtualnej
100	4	numer rejestru Y maszyny wirtualnej
101	5	etykieta, używana w funkcjach skoku
111	7	złożone wyrażenie (np. lista, liczba zmiennoprzecinkowa). Kompilator generuje wartość 4 dla tego tagu jeżeli argumentem jest złożone wyrażenie, znajdujące się w tablicy stałych. Indeks wyrażenia w tablicy stałych jest kolejnym argumentem zapisanym w postaci uniwersalnego indeksu. Jeżeli argumentem instrukcji jest lista (np. w przypadku instrukcji <code>select_val</code> ) to dla tego tagu generowana jest wartość 1. Kolejnymi bajtami w kodzie pośrednim są długość listy oraz jej elementy.

Tablica E.1: Tagi typów danych w pliku ze skompilowanym modulem

Jeżeli tagowana liczba jest nieujemna, mniejsza od 16 (możliwe jest zapisanie jej przy użyciu 4 bitów) to argument jest zapisany przy użyciu jednego bajtu a jego postać binarna to:

$$X_1X_2X_3X_4\mathbf{0TTT}_{(2)},$$

gdzie  $X_1X_2X_3X_4_{(2)}$  to tagowana liczba,  $X_1$  jest jej najbardziej znaczącym bitem, a  $TTT_{(2)}$  to tag danego typu argumentu.

Na przykład atom, który w tablicy atomów modułu ma indeks  $2_{10} = 10_2$ , po zakodowaniu będzie miał postać:

$$0010\mathbf{0010}_2 = 22_{16} = 34_{10}.$$

W przypadku gdy liczba jest nieujemna, mniejsza lub równa 16, a mniejsza od 2048 (możliwe jest jej zapisanie przy użyciu 11 bitów), argument jest zapisany przy użyciu dwóch bajtów, których postać binarna to:

$$X_1X_2X_3\mathbf{01TTT} X_4X_5X_6X_7X_8X_9X_{10}X_{11(2)},$$

gdzie  $X_1...X_{11(2)}$  to tagowana liczba,  $X_1$  jest jej najbardziej znaczącym bitem, a  $TTT_{(2)}$  to tag danego typu argumentu.

Na przykład, liczba całkowita  $565_{10} = 010\ 00110101_2$  po zakodowaniu będzie miała postać:

$$0100\mathbf{1001}\ 00110101_2 = 4935_{16} = 18741_{10}.$$

Jeżeli argument jest liczbą ujemną lub dodatnią wymagającą w zapisie dwójkowym więcej niż 11 bitów to liczba taka zapisywana jest binarnie w kodzie uzupełnień do dwóch (U2) poprzedzona odpowiednim nagłówkiem.



Jeżeli zakodowaną liczbę można zapisać na nie więcej niż 8 bajtach, to nagłówek ma następującą postać:

$$N_1N_2N_3\mathbf{11TTT}_{(2)},$$

gdzie  $N_1N_2N_3_{(2)}$  to rozmiar argumentu w bajtach pomniejszony o 2 (jeżeli argument jest liczbą ujemną zajmującą 1 bajt to powinien on zostać dopełniony do 2 bajtów),  $N_1$  jest jego najbardziej znaczącym bitem, a  $TTT_{(2)}$  to tag danego typu argumentu.

Na przykład, aby zapisać na dwóch bajtach liczbę  $-21_{10} = 11111111\ 11101011_{U2}$ , jej postać binarną należy poprzedzić nagłówkiem:

$$000\mathbf{11001}_2 = 19_{16} = 25_{10}.$$

Jeżeli do zapisania liczby w kodzie uzupełnień do dwóch potrzeba przynajmniej 9 bajtów, wtedy nagłówek jest dwubajtowy i ma postać:

$$1111\mathbf{1TTT}\ N_1N_2N_3N_4\mathbf{0000}_{(2)},$$

gdzie  $N_1N_2N_3N_4_{(2)}$  to rozmiar argumentu w bajtach pomniejszony o 9,  $N_1$  jest jego najbardziej znaczącym bitem, a  $TTT_{(2)}$  to tag danego typu argumentu.

Na przykład, w celu zapisania liczby  $2^{(15 \times 8) - 1} - 1$  na 15 bajtach, należy zapis tej liczby w kodzie U2 poprzedzić następującym nagłówkiem:

$$1111\mathbf{1001}\ 0110\mathbf{0000}_2 = F960_{16} = 63840_{10}.$$

## E.2. Lista instrukcji

W tabeli E.2 zawarto listę instrukcji rozumianych przez maszynę wirtualną BEAM wraz z jednobajtowym kodem operacji, listą jej argumentów i krótkim opisem działania.

Na liście oznaczono operacje, które zostały zaimplementowane w maszynie wirtualnej opisywanej w pracy. Brak implementacji poszczególnych instrukcji podyktowany jest brakiem wsparcia pewnych funkcjonalności lub typów danych w maszynie.

Instrukcje nieużywane przez kompilator Erlanga w wersji R16 zostały pominięte.

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
01	1	label Lbl	Wprowadza lokalną dla danego modułu etykietę identyfikującą aktualne miejsce w kodzie.	✓

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
02	2	func_info M F A	Definiuje funkcję F, w module M o arności A na jej początku. Instrukcja używana jest do generacji wyjątku <code>function_clause</code> dla funkcji, którą definiuje.	✓
03	3	int_code_end	Oznacza koniec kodu.	✓
04	4	call Arity Lbl	Wywołuje funkcję o arności Arity znajdującą się pod etykietą Lbl. Zapisuje następną instrukcję jako adres powrotu (wskaźnik <b>CP</b> ).	✓
05	5	call_last Arity Lbl Dest	Wywołuje rekurencyjną ogonowo funkcję o arności Arity znajdującą się pod etykietą Lbl. Nie zapisuje adresu powrotu. Przed wywołaniem zwalnia Dest+1 słów pamięci na stosie.	✓
06	6	call_only Arity Lbl	Wywołuje rekurencyjną ogonowo funkcję o arności Arity znajdującą się pod etykietą Lbl. Nie zapisuje adresu powrotu.	✓
07	7	call_ext Arity Dest	Wywołuje zewnętrzną funkcję o arności Arity mającą indeks Dest w tablicy funkcji zewnętrznych. Zapisuje następną instrukcję jako adres powrotu (wskaźnik <b>CP</b> ).	✓
08	8	call_ext_last Arity Des Dea	Wywołuje rekurencyjną ogonowo zewnętrzną funkcję o arności Arity mającą indeks Des w tablicy funkcji zewnętrznych. Nie zapisuje adresu powrotu. Przed wywołaniem zwalnia Dea+1 słów pamięci na stosie. Przywraca wskaźnik <b>CP</b> ze stosu.	✓
09	9	bif0 Bif Reg	Wywołuje wbudowaną funkcję Bif/0. Wynik zapisywany jest w rejestrze Reg.	✓
0A	10	bif1 Bif Arg Reg	Wywołuje wbudowaną funkcję Bif/1 z argumentem Arg. Wynik zapisywany jest w rejestrze Reg.	✓
0B	11	bif2 Bif Arg1 Arg2 Reg	Wywołuje wbudowaną funkcję Bif/2 z argumentami Arg1, Arg2. Wynik zapisywany jest w rejestrze Reg.	✓

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
0C	12	allocate StackN Live	Alokuje miejsce dla StackN słów na stosie. Używanych jest Live rejestrów <b>X</b> , gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> . Zapisuje <b>CP</b> na stosie.	✓
0D	13	allocate_heap StackN HeapN Live	Alokuje miejsce dla StackN słów na stosie. Upewnia się że na sterpie jest HeapN wolnych słów. Używanych jest Live rejestrów <b>X</b> , gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> . Zapisuje aktualną wartość <b>CP</b> na stosie.	✓
0E	14	allocate_zero StackN Live	Tak jak allocate/2, ale zaalokowana pamięć jest wyzerowana.	✓
0F	15	allocate_heap_zero SN HN L	Tak jak allocate_heap/3, ale zaalokowana pamięć jest wyzerowana.	✗
10	16	test_heap HN L	Upewnia się że na sterpie jest HN wolnych słów. Używanych jest L rejestrów <b>X</b> , gdyby w trakcie konieczne było uruchomienie <i>garbage collector</i> .	✓
11	17	init N	Zeruje N-te słowo na stosie. Instrukcja poprzednio nazywała się <i>kill</i> i jako taka może jeszcze występować w pewnych miejscach.	✓
12	18	deallocate N	Przywraca <b>CP</b> ze stosu i dealokuje N+1 słów ze stosu.	✓
13	19	return	Wraca do adresu zapisanego we wskaźniku <b>CP</b> .	✓
14	20	send	Wysyła wiadomość z rejestru <b>X1</b> do procesu w rejestrze <b>X0</b> .	✓
15	21	remove_message	Usuwa aktualną wiadomość z kolejki wiadomości. Zapisuje wskaźnik do niej w rejestrze <b>X0</b> . Usuwa aktywne przeterminowanie ( <i>timeout</i> ).	✓

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
16	22	timeout	Resetuje wskaźnik kolejnej wiadomości do odczytania na początek kolejki wiadomości. Czyści flagę przeterminowania.	✓
17	23	loop_rec Lbl Src	Zapisuje kolejną wiadomość w kolejce wiadomości Src w rejestrze <b>R0</b> . Jeśli jest pusta wykonuje skok do etykiety Lbl.	✓
18	24	loop_rec_end Lbl	Ustawia wskaźnik zapamiętujący kolejną wiadomość do odczytania na kolejną wiadomość w kolejce wiadomości i wykonuje skok do etykiety Lbl.	✓
19	25	wait Lbl	Zawiesza proces aż do otrzymania wiadomości, który zostanie wznowiony na początku bloku receive w etykiecie Lbl.	✓
1A	26	wait_timeout Lbl T	Zawiesza proces jak wait. Ustawia przeterminowanie T i zapisuje następną instrukcję, która zostanie wykonana jeśli przeterminowanie się zrealizuje.	✓
27	39	is_lt Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest większe lub równe od Arg2.	✓
28	40	is_ge Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest mniejsze Arg2.	✗
29	41	is_eq Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest arytmetycznie różne od Arg2.	✗
2A	42	is_ne Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest arytmetycznie równe Arg2.	✗
2B	43	is_eq_exact Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest różne Arg2.	✓
2C	44	is_ne_exact Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest równe Arg2.	✓
2D	45	is_integer Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on liczbą całkowitą.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
2E	46	is_float Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on liczbą rzeczywistą.	✗
2F	47	is_number Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on liczbą.	✗
30	48	is_atom Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on atomem.	✓
31	49	is_pid Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on identyfikatorem procesu.	✗
32	50	is_reference Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on referencją.	✗
33	51	is_port Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on portem.	✗
34	52	is_nil Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on znacznikiem końca listy (NIL).	✓
35	53	is_binary Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on zmienną binarną.	✗
37	55	is_list Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on ani listą ani znacznikiem końca listy.	✗
38	56	is_nonempty_list Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on niepustą listą.	✓
39	57	is_tuple Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on krotką.	✓
3A	58	test_arity Lbl Arg1 Arity	Sprawdza arność krotki Arg1 i skacze do Lbl jeśli nie jest ona równa Arity.	✓
3B	59	select_val Arg Lbl Dest	Skacze do etykiety Dest [Arg]. Jeśli nie istnieje skacze do Lbl.	✓
3C	60	select_tuple_arity Tuple Lbl Dest	Sprawdza arność krotki Tuple i skacze do etykiety Dest [Arity]. Jeśli etykieta nie istnieje skacze do Lbl.	✓
3D	61	jump Lbl	Skacze do etykiety Lbl.	✓
3E	62	catch Dest Lbl	Tworzy nowy blok catch. Zapisuje etykietę Lbl na Dest miejscu na stosie.	✗
3F	63	catch_end Dest	Kończy blok catch. Wymazuje etykietę na miejscu Dest na stosie.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
40	64	move Src Dest	Przenosi wartość z Src do rejestru Dest.	✓
41	65	get_list Src Hd Tail	Umieszcza głowę listy Src w rejestrze Hd i jej ogon w rejestrze Tail.	✓
42	66	get_tuple_element Src Elem Dest	Umieszcza element Elem krotki Src w rejestrze Dest.	✓
43	67	set_tuple_element Elem Tuple Pos	Umieszcza element Elem w krotce Tuple na pozycji Pos.	✗
45	69	put_list Hd Tail Dest	Tworzy komórkę listy [Hd Tail] na szczycie sterty i umieszcza ją w rejestrze Dest.	✓
46	70	put_tuple Dest Arity	Tworzy krotkę o arności Arity na szczycie sterty i umieszcza ją w rejestrze Dest.	✓
47	71	put Arg	Umieszcza Arg na szczycie sterty.	✓
48	72	badmatch Arg	Rzuca wyjątek badmatch z argumentem Arg.	✓
49	73	if_end	Rzuca wyjątek if_clause.	✓
4A	74	case_end Arg	Rzuca wyjątek case_clause z argumentem Arg.	✓
4B	75	call_fun Arity	Woła obiekt funkcyjny o arności Arity. Zakłada, że argumenty znajdują się w rejestrach <b>X0...X(Arity-1)</b> , a lambda w rejestrze <b>X(Arity)</b> . Zapisuje następną instrukcję we wskaźniku <b>CP</b> .	✗
4D	77	is_function Lbl Arg1	Sprawdza typ argumentu Arg1 i skacze do Lbl jeśli nie jest on funkcją.	✗
4E	78	call_ext_only Arity Lbl	Wywołuje rekurencyjną ogonowo zewnętrzną funkcję o arności Arity mającą indeks Lbl w tablicy funkcji zewnętrznych. Nie zapisuje adresu powrotu.	✓

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
59	89	bs_put_integer Fail Size Unit Flags Src	Umieszcza liczbę całkowitą w utworzonym wcześniej kontekście zmiennej binarnej, zajmując w nim rozmiar Size w jednostkach Unit. Źródłowa liczba znajduje się w rejestrze Src. Flagi dotyczące sposobu umieszczenia zmiennej w kontekście znajdują się w liczbie całkowitej Flags. Adres skoku Fail jest nieużywany.	✗
5A	90	bs_put_binary Fail Size Unit Flags Src	Umieszcza zmienną binarną w utworzonym wcześniej kontekście zmiennej binarnej, zajmując w nim rozmiar Size w jednostkach Unit. Źródłowa zmienna znajduje się w rejestrze Src. Flagi dotyczące sposobu umieszczenia zmiennej w kontekście znajdują się w liczbie całkowitej Flags. Adres skoku Fail jest nieużywany.	✗
5B	91	bs_put_float Fail Size Unit Flags Src	Umieszcza liczbę zmiennoprzecinkową w utworzonym wcześniej kontekście zmiennej binarnej, zajmując w nim rozmiar Size w jednostkach Unit. Źródłowa liczba znajduje się w rejestrze Src. Flagi dotyczące sposobu umieszczenia liczby zmiennoprzecinkowej w kontekście znajdują się w liczbie całkowitej Flags. Adres skoku Fail jest nieużywany.	✗
5C	92	bs_put_string Size Bytes	Bezpośrednio umieszcza bajty w utworzonym wcześniej kontekście zmiennej binarnej. Źródłowe bajty wskazywane są przez wskaźnik Bytes a ich liczba do przekopowania to Size.	✗
5E	94	fclearerror	Czyści flagę błędu zmiennoprzecinkowego, jeśli jest ustawiona.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
5F	95	fcheckerror Arg0	Sprawdza czy Arg0 zawiera wartość NaN lub nieskończoność. Jeśli tak, rzuca wyjątek badarith.	✗
60	96	fmove Arg0 Arg1	Kopiuje wartość Arg0 do Arg1.	✗
61	97	fconv Arg0 Arg1	Konwertuje wartość spod Arg0 na liczbę zmiennoprzecinkową i umieszcza ją w rejestrze Arg1.	✗
62	98	fadd Arg0 Arg1 Arg2 Arg3	Zapisuje w rejestrze Arg3 wynik dodawania Arg1 do Arg2. Argument Arg0 jest nieużywany.	✗
63	99	fsub Arg0 Arg1 Arg2 Arg3	Zapisuje w rejestrze Arg3 wynik odejmowania Arg2 od Arg1. Argument Arg0 jest nieużywany.	✗
64	100	fmul Arg0 Arg1 Arg2 Arg3	Zapisuje w rejestrze Arg3 wynik mnożenia Arg1 przez Arg2. Argument Arg0 jest nieużywany.	✗
65	101	fdiv Arg0 Arg1 Arg2 Arg3	Zapisuje w rejestrze Arg3 wynik dzielenia Arg1 przez Arg2. Argument Arg0 jest nieużywany.	✗
66	102	fnegate Arg0 Arg1 Arg2	Zapisuje ujemną wartość z rejestru Arg1 w rejestrze Arg2. Argument Arg0 jest nieużywany.	✗
67	103	make_fun2 N	Odczytuje wpis o indeksie N w tablicy lambd modułu i umieszcza go w rejestrze X0.	✗
68	104	try Dest Label	Tak jak instrukcja catch/2.	✗
69	105	try_end Dest	Kończy blok catch. Wymazuje etykietę na miejscu Dest na stosie. Jeśli nie ma zapisanej wartości w rejestrze R0 oznacza to że w bloku catch złapano wyjątek. W takim przypadku dokonywane jest przepisanie wartości rejestrów: R0 = X1, X1 = X2 i X2 = X3.	✗
6A	106	try_case Dest	Jak instrukcja try_end/1.	✗



Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
6B	107	try_case_end Reason	Rzuca wyjątek try_clause z argumentem Reason.	✗
6C	108	raise Stacktrace Reason	Rzuca wyjątek Reason ze stosem wywołań Stacktrace.	✗
6D	109	bs_init2 Fail Sz Words Regs Flags Dst	Inicjalizuje miejsce dla zmiennej binarnej o rozmiarze Sz. Zapewnia, że oprócz tego rozmiaru po inicjalizacji dostępne będzie dodatkowo Words słów maszynowych. Regs oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie inicjalizacji konieczne było uruchomienie <i>garbage collector</i> . Adres skoku Fail oraz opcje Flags są aktualnie nieużywane.	✗
6F	111	bs_add Fail S1 S2 Unit Dst	Oblicza sumę bitów w S1 i liczby jednostek Unit w S2. Wynik przechowuje w Dst. W przypadku niepowodzenia wykonuje skok do etykiety Fail.	✗
70	112	apply N	Znajduje adres początku funkcji zapisanej w rejestrze $X(N+1)$ , w module zapisanym w rejestrze $X(N)$ o arności N i skacze do tego adresu. Zapisuje następną instrukcję we wskaźniku CP.	✓
71	113	apply_last N Dea	Skacze do zewnętrznej funkcji tak jak instrukcja apply/1. Ściąga wartość wskaźnika CP ze stosu. Zwalnia Dea miejsc na szczycie stosu.	✗
72	114	is_boolean/2	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on ani atomem true ani false.	✗
73	115	is_function2 Lbl Arg1 Arity	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on funkcją o arności Arity.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
74	116	bs_start_match2 Fail Bin X Y Dst	Sprawdza czy Bin jest kontekstem zmiennej binarnej z odpowiednią liczbą miejsc do zapisania tymczasowych przesunięć bitowych ( <i>offsetów</i> ) określonych przez liczbę Y. Jeśli aktualna liczba miejsc jest za mała tworzony jest nowy kontekst porównywania zmiennej binarnej z odpowiednią liczbą miejsc, który zostanie zapisany w miejscu Dst. W przypadku niepowodzenia dokonany zostanie skok do etykiety Fail. X oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> a.	✗
75	117	bs_get_integer2 Fail Ms Live Sz Unit Flags Dst	Pobiera liczbę całkowitą z kontekstu zmiennej binarnej Ms o rozmiarze Sz w jednostkach Unit. Wynik zapisywany jest do Dst. Opcje operacji zapisywane są w liczbie całkowitej Flags. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> a.	✗
76	118	bs_get_float2 Fail Ms Live Sz Unit Flags Dst	Pobiera liczbę zmiennoprzecinkową z kontekstu zmiennej binarnej Ms o rozmiarze Sz w jednostkach Unit. Wynik zapisywany jest do Dst. Opcje operacji zapisywane są w liczbie całkowitej Flags. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> a.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
77	119	bs_get_binary2 Fail Ms Live Sz Unit Flags Dst	Pobiera zmienną binarną z kontekstu zmiennej binarnej Ms o rozmiarze Sz w jednostkach Unit. Wynik zapisywany jest do Dst. Opcje operacji zapisywane są w liczbie całkowitej Flags. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> .	✗
78	120	bs_skip_bits2 Fail Ms Sz Unit Flags	Pomija Sz jednostek wyrażonych w Unit z kontekstu zmiennej binarnej Ms. Opcje operacji wyrażone są za pomocą liczby całkowitej Flags. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail.	✗
79	121	bs_test_tail2 Fail Ms Bits	Sprawdza, czy kontekst zmiennej binarnej Ms ma jeszcze dokładnie Bits niedopasowanych bitów. Wykonuje skok do etykiety Fail jeżeli tak nie jest.	✗
7A	122	bs_save2 Reg Index	Zapisuje aktualne przesunięcie bitowe <i>offset</i> z kontekstu zmiennej binarnej zawartego w rejestrze Reg i zapisuje do jego tablicy <i>offsetów</i> .	✗
7B	123	bs_restore2 Reg Index	Odczytuje przesunięcie bitowe kontekstu zmiennej binarnej zapisanego w rejestrze Reg z indeksu Index. Zapisuje go jako aktualne dla tego kontekstu.	✗
7C	124	gc_bif1 Lbl Live Bif Arg1 Reg	Wywołuje funkcję wbudowaną Bif/1 z argumentem Arg1. Wynik zapisuje w rejestrze Reg. W przypadku niepowodzenia skacze do etykiety Lbl. Uruchamia <i>garbage collector</i> jeśli jest to konieczne, zachowując Live rejestrów X.	✓

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
7D	125	gc_bif2 Lbl Live Bif Arg1 Arg2 Reg	Wywołuje funkcję wbudowaną Bif/2 z argumentami Arg1, Arg2. Wynik zapisuje w rejestrze Reg. W przypadku niepowodzenia skacze do etykiety Lbl. Uruchamia <i>garbage collector</i> jeśli jest to konieczne, zachowując Live rejestrów <b>X</b> .	✓
81	129	is_bitstr Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on ciągiem bitów.	✗
82	130	bs_context_to_binary Reg	Zamienia kontekst zmiennej binarnej znajdującej się w rejestrze Reg na właściwą zmienną binarną.	✗
83	131	bs_test_unit Fail Ms Unit	Sprawdza czy rozmiar niedopasowanego jeszcze fragmentu kontekstu zmiennej binarnej Ms jest podzielny przez Unit. Jeżeli nie, wykonywany jest skok do etykiety Fail.	✗
84	132	bs_match_string Fail Ms Bits Val	Dokonyuje porównania Bits bitów, począwszy od miejsca wskazywanego przez wskaźnik Val z kontekstem zmiennej binarnej Ms. Jeżeli porównywane wartości nie są równe dokonywany jest skok do etykiety Fail.	✗
85	133	bs_init_writable	Alokuje miejsce o rozmiarze <b>R0</b> na stercie procesu. Tworzy w zaalokowanym miejscu strukturę zmiennej binarnej. Dodatkowo tworzy wskaźnik do utworzonej struktury, który umieszczony zostanie w rejestrze <b>R0</b> .	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
86	134	bs_append Fail Size Extra Live Unit Bin Flags Dst	Dopisuje Size jednostek Unit do zmiennej binarnej Bin i zapisuje wynik do Dst. Jeśli nie ma wystarczająco dużej ilości miejsca, tworzona jest nowa struktura na sterpie z odpowiednią ilością miejsca, powiększona dodatkowo o Extra słów maszynowych. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail. Opcje operacji zapisane są w liczbie całkowitej Flags. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> .	✗
87	135	bs_private_append Fail Size Unit Bin Flags Dst	Operacja ma działanie podobne do operacji bs_append/8 jednak w przypadku zbyt małej ilości miejsca dokonuje realokacji aktualnej zmiennej.	✗
88	136	trim N Remaining	Redukuje stos o N słów, zachowując CP na jego szczycie.	✓
89	137	bs_init_bits Fail Sz Words Regs Flags Dst	Alokuje zmienną binarną na sterpie o rozmiarze Sz bitów. Jeżeli rozmiar nie jest podzielny przez 8, tworzony jest wskaźnik na strukturę z zapisaną ilością zajmowanych bitów. Wynik operacji zapisany jest do Dst. Upewnia się że na sterpie jest dodatkowo Words słów maszynowych możliwych do zaalokowania. Opcje operacji zapisane są w liczbie całkowitej Flags. W razie niepowodzenia wykonywany jest skok do etykiety Fail. Regs oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> .	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
8A	138	bs_get_utf8 Fail Ms Arg2 Arg3 Dst	Pobiera znak zapisany w UTF-8 z kontekstu zmiennej binarnej Ms i zapisuje go do Dst. W przypadku niepowodzenia wykonuje skok do etykiety Fail. Argumenty Arg2 oraz Arg3 są aktualnie nieużywane.	✗
8B	139	bs_skip_utf8 Fail Ms Arg2 Arg3	Pomija znak zakodowany w UTF-8 w kontekście zmiennej binarnej Ms. W przypadku niepowodzenia wykonuje skok do etykiety Fail. Argumenty Arg2 oraz Arg3 są aktualnie nieużywane.	✗
8C	140	bs_get_utf16 Fail Ms Arg2 Flags Dst	Pobiera znak zapisany w UTF-16 z kontekstu zmiennej binarnej Ms, używając opcji zapisanych w liczbie całkowitej Flags i zapisuje go do Dst. W przypadku niepowodzenia wykonuje skok do etykiety Fail. Argument Arg2 jest aktualnie nieużywany.	✗
8D	141	bs_skip_utf16 Fail Ms Arg2 Flags	Pomija znak zakodowany w UTF-16 w kontekście zmiennej binarnej Ms, używając opcji zapisanych w liczbie całkowitej Flags. W przypadku niepowodzenia wykonuje skok do etykiety Fail. Argument Arg2 jest aktualnie nieużywany.	✗
8E	142	bs_get_utf32 Fail Ms Live Flags Dst	Pobiera znak zapisany w UTF-32 z kontekstu zmiennej binarnej Ms, używając opcji zapisanych w liczbie całkowitej Flags i zapisuje go do Dst. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> . W przypadku niepowodzenia wykonuje skok do etykiety Fail.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
8F	143	bs_skip_utf32 Fail Ms Live Flags	Pomija znak zakodowany w UTF-32 w kontekście zmiennej binarnej Ms, używając opcji zapisanych w liczbie całkowitej Flags. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> . W przypadku niepowodzenia wykonuje skok do etykiety Fail.	✗
90	144	bs_utf8_size Fail Literal Dst	Oblicza liczbę bajtów koniecznych do zapisania Literal w UTF-8 i zapisuje wynik do Dst. Argument Fail jest aktualnie nieużywany.	✗
91	145	bs_put_utf8 Fail Flags Src	Umieszcza znak zakodowany w UTF-8 znajdujący się w Src w aktualnym kontekście zmiennej binarnej. Argumenty Fail i Flags są aktualnie nieużywane.	✗
92	146	bs_utf16_size Fail Literal Dst	Oblicza liczbę bajtów koniecznych do zapisania Literal w UTF-16 i zapisuje wynik do Dst. Argument Fail jest aktualnie nieużywany.	✗
93	147	bs_put_utf16 Fail Flags Literal	Umieszcza znak zakodowany w UTF-16 znajdujący się w Src w aktualnym kontekście zmiennej binarnej z użyciem opcji zapisanych w liczbie całkowitej Flags. Argument Fail jest aktualnie nieużywany.	✗
94	148	bs_put_utf32 Fail Flags Literal	Umieszcza znak zakodowany w UTF-32 znajdujący się w Src w aktualnym kontekście zmiennej binarnej z użyciem opcji zapisanych w liczbie całkowitej Flags. W przypadku niepowodzenia wykonuje skok do etykiety Fail.	✗
95	149	on_load	Oznacza kod wykonywany przy ładowaniu modułu.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
96	150	recv_mark Lbl	Zapamiętuje aktualną wiadomość z kolejki oraz etykietę Label do instrukcji loop_rec/2.	<b>X</b>
97	151	recv_set Lbl	Jeśli etykieta Lbl wskazuje na instrukcję loop_rec/2 to przepisuje wiadomość zachowaną przez instrukcję recv_mark do wskaźnika kolejnej wiadomości do odczytania.	<b>X</b>
98	152	gc_bif3 Lbl Live Bif Arg1 Arg2 Arg3 Reg	Wywołuje funkcję wbudowaną Bif/3 z argumentami Arg1, Arg2, Arg3. Wynik zapisuje w rejestrze Reg. W przypadku niepowodzenia skacze do etykiety Lbl. Uruchamia <i>garbage collector</i> jeśli jest to konieczne, zachowując Live rejestrów <b>X</b> .	<b>X</b>
99	153	line N	Znakuje aktualne miejsce jako linia o indeksie N w tablicy linii.	<b>X</b>

Tablica E.2: Lista operacji maszyny wirtualnej BEAM



## Bibliografia

- [1] Ericsson AB. Erlang Embedded Systems User's Guide, 1997.
- [2] 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].
- [3] 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].
- [4] 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].
- [5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [6] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, 2003.
- [7] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd., 2011.
- [8] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.
- [9] HopeRF Electronic. *RFM73 Datasheet*. Hope Microelectronics Co. Ltd., 2006.
- [10] Anton Ertl. Threaded Code. <http://www.complang.tuwien.ac.at/forth/threaded-code.html>, 2004. [data dostępu: 12.07.2014].
- [11] Jim Gray. *Why Do Computers Stop And What Can Be Done About It?*, 1985.
- [12] 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.
- [13] Maxim Kharchenko. Erlang on Xen, A quest to lower startup latency. *Erlang Factory SF Bay Area 2012, San Francisco*, 2012.

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