ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1993

Seria: INFORMATYKA z. 22 Nr kol. 1195

Dariusz ZONENBERG
Instytut Informatyki Teoretycznej i Stosowanej
Polskiej Akademii Nauk

EMULATOR UKŁADOWY — OBSERWACJA DANYCH DO ANALIZY ZACHOWAŃ SYSTEMÓW CZASU RZECZYWISTEGO

Streszczenie. W opracowaniu przedstawiono metody uruchamiania oprogramowania systemów mikroprocesorowych pod kątem struktur danych rezydujących w pamięci operacyjnej. Zostały przedstawione przeprowadzane za pomocą emulatora układowego metody detekcji zmian w pamięci, powstałych na skutek działania programu. Jako konkluzja został zaproponowany przykładowy, wzorowany na języku *Pascal*, język opisu procesu śledzenia i prezentacji danych systemu mikroprocesorowego, uruchamianego za pomocą emulatora układowego.

IN-CIRCUIT EMULATOR — DATA OBSERVATION FOR BEHAVIORAL ANALYSIS OF REAL TIME SYSTEMS

Summary. The article presents microprocessor system software debugging techniques taking into account data structures residing in operating memory. Presented are methods of detecting of data modifications in operating memory during program execution that can be performed using an ICE (In-Circuit Emulator). The conclusion of the paper is an example *Pascal*-like language for describing the tracing and presentation of the data of the microprocessor system debugged with an ICE.

EMULATEUR MATERIEL — OBSERVATION DES DONNEES POUR L'ANALYSE DES COMPORTEMENTS DE SYSTEMES TEMPS REEL

Resumé. Nous presentons les méthodes de mettre au point un logiciel conçu pour les systèmes microprocesseurs, compte tenu des structures de données résidentes en mémoire interne. Les méthodes de détection de changements de mémoire, faits par les programmes sont discutées. Dans la suite un langage de type *Pascal* est proposé pour la description du processus d'observation et présentation des données dans un système microprocesseur étant mis en marche à l'aide de l'emulateur matériel.

1. Wprowadzenie

Jedną z głównych dziedzin zastosowania mikroprocesorów jest konstrukcja sterowników, używanych zarówno w przemyśle, jak i przedmiotach codziennego użytku. Jako oprogramowanie tego typu urządzeń używa się często małych wielozadaniowych systemów czasu rzeczywistego. Pozwalają one na szybkie tworzenie oprogramowania sterownika pracującego w czasie rzeczywistym, łatwą pielęgnację uprzednio zdekomponowanego na wiele zadań oprogramowania oraz możliwość prostego oszacowania czasów reakcji sterownika na zmiany w otoczeniu. Niedogodnością w konstruowaniu oprogramowania tej klasy (niekoniecznie opartego na jądrze wielozadaniowym) jest konieczność stosowania złożonych technik uruchamiania w czasie rzeczywistym, kiedy to typowe programy uruchamiające lub symulatory procesorów nie są w stanie sprostać wymogom programisty. Sytuacja ta powoduje konieczność tworzenia złożonych narzędzi uruchomieniowych, umożliwiających z jednej strony efektywne śledzenie realizacji oprogramowania czasu rzeczywistego, a z drugiej wstępną obróbkę śledzonych zdarzeń oraz ich ergonomiczną prezentację. Jednym z podstawowych obszarów dla wspomnianych działań jest obserwacja i prezentacja danych.

Spośród współczesnych środków uruchamiania oprogramowania czasu rzeczywistego do najbardziej efektywnych należą emulatory układowe mikroprocesorów, pełniące funkcję zarówno debuggerów, jak i analizatorów magistrali. Opisy ich możliwości funkcjonalnych można znaleźć w pozycjach [1], [2], [3], [4] oraz [5].

Obecnie gdy oprogramowanie czasu rzeczywistego zwykle jest oparte na systemach czasu rzeczywistego oraz często pisane jest w językach wyższego poziomu, jak *Ada*, *Modula-2*, *C* lub języki specjalizowane, coraz większej wagi nabiera obserwacja złożonych struktur danych tego oprogramowania. Przykładowe, rozwinięte rozwiązania tego

problemu dla debuggera systemu MS-DOS napotyka się w ostatnich wersjach debuggerów zarówno firmy *Microsoft* [6], jak i innych producentów oprogramowania narzędziowego.

W przypadku emulatorów układowych możliwość obserwacji otoczenia logicznego mikroprocesora jest rozwiązywana jako gotowy zestaw zleceń wyświetlających stan zasobów [2] lub jako zestaw makroinstrukcji, umożliwiających tworzenie krótkich programów wyświetlających informację w żądanym formacie według bieżącego stanu mikroprocesora. To drugie rozwiązanie napotykamy w produktach firmy *Intel* ([1], [3]) i na nim był również wzorowany język emulatora RTDS/2-PC, który będzie użyty w poniższym tekście do ilustracji przykładów [5].

W celu wyłącznej wizualizacji danych przydatne są również emulatory pamięci, pozwalające na dostęp tylko do pamięci operacyjnej procesora, ale ze względu na ograniczony zakres ich stosowania nie będą one w poniższym tekście analizowane.

2. Oprogramowanie czasu rzeczywistego

Dla większości procesorów prawdziwe jest założenie, że znając stan zasobów, do których procesor ma dostęp, można w sposób deterministyczny określić jego zachowanie w kolejnych i przyszłych przedziałach czasowych. Zasobami są:

- pamięć kodu programu procesora;
- pamięć danych programu;
- rejestry procesora;
- łącze maszyny cyfrowej z jej otoczeniem (porty, urządzenia zewnętrzne itp.).

Niezależnie od operacji wykonywanych przez procesor, zawsze mogą zmienić się dane na łączu z otoczenia i zawsze można wyznaczyć skończony okres, w którym są one ustalone. W takim przypadku należy założyć, że prawidłowo przygotowany kod oprogramowania zawiera w sobie określoną, deterministyczną reakcję na tego typu zdarzenie. Zatem przewidywalność zachowań procesora w stosunku do danych z otoczenia zawiera się, w istocie, w poprawności kodu programu. Opierając się na takich przesłankach, można sformułować kilka poniższych założeń:

1. Stan procesora, realizującego dane oprogramowanie, określa zbiór wszystkich przyszłych deterministycznych zachowań, jednoznacznie wyznaczonych przez kod, dane i rejestry procesora.

2. Stan procesora realizującego dane oprogramowanie będzie odtąd nazywany stanem oprogramowania, zgodnie z punktem widzenia programisty, że projektowane i uruchamiane jest oprogramowanie, a sam procesor jest co prawda nieodzowny, ale niezależny od programisty.

3. Zgodnie z powyższymi punktami stan oprogramowania jest jednoznacznie określony przez kod programu, jego dane i rejestry procesora.

Poniższe opracowanie skupia się na mikroprocesorach, dla których używane są emulatory układowe w celu uruchomienia oprogramowania w warunkach pracy w realnym otoczeniu i rzeczywistych uzależnieniach czasowych. Są to z reguły procesory o stosunkowo prostej architekturze, zgodnej z modelem procesora zaproponowanym przez von Neumana (np. rodziny 8048, 8051, 8080, 80(x)86, 80(x)96, 6502, 6800, 680x0 i wiele innych), zatem powyższe definicje dotyczą również tej klasy procesorów.

W określeniu stanu systemu przetwarzającego dane jako oprogramowania sterującego danym kontrolerem wystarczające są następujące zasoby:

- pamięć zawierająca oprogramowanie;
- dane w pamięci operacyjnej;
- zawartość rejestrów procesora (w tym licznika programu).

Zawartość pamięci programu jest z reguły stała i znana. Programista uruchamiający oprogramowanie jest z reguły zainteresowany tym, w jaki sposób jest wykonywany jego program i informację o tym uzyskuje na podstawie zmian zachodzących w danych (pamięć operacyjna i rejestry).

Analizując pracę systemu operacyjnego programista przekonuje się, że większość czasu system spędza na oczekiwaniu na określone zdarzenia. Część czasu systemu jest zużywana na operacje związane z obsługą zegara czasu rzeczywistego, a w czasie przetwarzania i przesyłania danych sterowanie często przechodzi do jądra systemu. Jego struktura może być nie znana, jeśli wykorzystywany jest jakiś system komercjalny. Powoduje to, że często miejsce, w którym znajduje się realizowany przez procesor kod, niewiele mówi o stanie całego oprogramowania. Z kolei śledzenie poszczególnych cykli realizacji kodu zawiera zbyt wiele nieistotnych dla programisty informacji, utrudniających tylko rozeznanie stanu oprogramowania.

W jednoprocesorowych systemach operacyjnych (a taki tu analizujemy) każdy proces posiada własny zestaw rejestrów, przechowywany z reguły w pamięci operacyjnej, jeśli dany proces nie jest aktywny. Zakładając, iż:

1. oprogramowanie używa rejestrów tylko do przetwarzania danych, a wyniki przetwarzania zawsze przechowywane są w pamięci,

2. procesor posiada tylko jeden prosty zestaw rejestrów, więc jeśli w środowisku systemu operacyjnego utworzono kilka procesów, to wszystkie ich dane, łącznie z zawartością rejestrów rezydują w pamięci operacyjnej (co istotnie ma miejsce dla większości procesorów i systemów),

możemy wysunąć wniosek, że śledząc zmiany zachodzące w danych ulokowanych w pamięci operacyjnej programista może wyciągnąć najważniejsze wnioski dotyczące sposobu przetwarzania informacji przez oprogramowanie. Istotnie — doświadczenia z systemami operacyjnymi dla procesorów popularnych w Polsce rodzin 8080/85/Z80 i 8086 oraz analiza procesorów rodzin 6800, 68000, 6502 skłania do przyjęcia powyższego wniosku, chociaż dla ich wersji z pamięcią notatnikową wewnątrz procesora mogą następować opóźnienia pomiędzy zmianą danej w pamięci, a rzeczywistym dostępem do pamięci na zewnątrz procesora. Nie jest on natomiast prawdziwy dla wielu mikrokomputerów jednoukładowych — mają one często proste możliwości, ale rozbudowaną strukturę rejestrów, co czasem powoduje, że pamięć operacyjna na zewnątrz procesora jest po prostu zbędna.

W strukturach znanych systemów operacyjnych czasu rzeczywistego można wyróżnić kilka klas danych, które są z reguły przechowywane w pamięci. Ich szczegółowa realizacja zależy od konkretnego systemu operacyjnego, zostaną więc one podane tylko ogólnie, aby dać pogląd na charakter obiektów w pamięci operacyjnej procesora. Przyjmijmy zatem następujący ich podział:

1. Deskryptory procesów oraz dane organizacyjne procesów (zawartość rejestrów, stan procesów, status komunikacji z jądrem systemu).

2. Dane lokalne procesów — statyczne i dynamiczne (stos).

3. Bufory komunikatów, za pomocą których następuje wymiana informacji pomiędzy procesami — zarządza nimi system operacyjny.

4. Semafory zapewniające synchronizację poszczególnych procesów.

5. Kolejki procesów, oczekujących na określone zdarzenie, i komunikatów.

6. Dane ewentualych procedur komunikacyjnych (np. konsoli, dysków, sieci) — w pewnych systemach (np. wzorowanych na jądrze systemu UNIX) nie związane z żadnym procesem, a istotne dla stanu oprogramowania.

Wszystkie powyższe dane są zwykle ulokowane w pamięci operacyjnej mikroprocesora, a ich szczegółowa realizacja zależy od systemu operacyjnego. Dla małych systemów operacyjnych czasu rzeczywistego nie wszystkie elementy muszą być zaimplementowane.

3. Narzędzia analizy i dostępu do danych

Posiadając narzędzie w postaci emulatora układowego lub emulatora pamięci operacyjnej programista, wykorzystujący je w celach uruchomieniowych, może w określony sposób śledzić zmiany zachodzące w danych procesora. Nasuwają się do wykorzystania następujące ogólne metody śledzenia dostępu do danych:

1. Sprzętowa detekcja poszczególnych cykli magistrali procesora za pomocą emulatora układowego. Analiza cykli zapisu pod kątem dostępu do pamięci w interesujących użytkownika obszarach.

2. Deklaracja przez użytkownika pamięci emulowanej oraz sprzętowa detekcja zapisu do interesujących obszarów pamięci.

3. Detekcja rozkazów przydzielających i zwalniających pamięć danych dynamicznych (stos) na potrzeby programu, umożliwiająca śledzenie lokalnych danych dynamicznych. Detekcja dostępu do tak wyliczonego obszaru danych dynamicznych może być realizowana metodą 1 lub 2.

Przyjmijmy, uniezależniając się od konkretnej realizacji sprzętowej, że start detekcji następuje w emulatorze po wywołaniu procedury o następującyn nagłówku:

```
procedure DetectDataWrite(Relation : string);

(* Gdzie parametr Relation zawiera lancuch okreslajacy za
pomoca umownej zmiennej  $ADDRESS, jaki obszar adresowy
podlega detekcji zapisu, np.:

1) '$ADDRESS >= 0 AND $ADDRESS <= $FFF' - przedzial od 0 do FFF(hex)

2) '($ADDRESS >= 0 AND $ADDRESS <= $3FF) OR $ADDRESS > $FE000' -
   przedzial od 0 do 3FF(hex) lub adresy wieksze od FE000(hex)

*)
```

W konkretnej realizacji emulatora, przykładowo RTDS/2–PC, powyższe relacje będą miały odpowiednio następującą postać:

```
1) 1C wrm irg 0:0 0:FFF any

2) 1C wrm irg 0:0 0:3FF any
   2C wrm at> F000:E000 any
```

Dotychczas opisano sposoby sprawdzania, czy dane systemu nie uległy zmianie. Kolejnym ważnym zadaniem jest dostęp do nich od strony emulatora i ich wizualizacja.

Głównym problemem dostępu do danych jest konieczność zachowania trybu pracy oprogramowania w czasie rzeczywistym. Wiąże się to z aktualnością danych; podczas pobierania danych otoczenie uruchamianego prototypu może się zmienić, a dostęp do danych od strony emulatora powinien zapewnić odczyt jej nowej wartości przed kolejną jej zmianą. Można pokazać, że prawdopodobna jest sytuacja, gdy dane zmieniają się w pewnych sytuacjach tak szybko, że emulator musi mieć wyłączność w dostępie do nich, aby wychwycić zmiany. Z drugiej strony wyłączność w dostępie może wstrzymywać pracę emulowanego systemu.

Drugim problemem może być zbyt duża częstość zmian w danych, powodująca niemożność nadążenia z wizualizacją przez oprogramowanie emulatora. Spowoduje to przepełnienie buforów pamiętających kolejne zmiany lub utratę informacji o części zmian.

Kolejne znane metody dostępu do danych emulowanego prototypu są następujące:

a) BRAK DOSTĘPU W CZASIE RZECZYWISTYM — Dostęp poprzez kanał komunikacyjny emulatora układowego. Metoda ta angażuje emulowany procesor — istnieje więc niebezpieczeństwo, że otoczenie procesora zmieni się w czasie dostępu do danych. Konieczne jest zamrożenie pracy emulowanego prototypu, a nie zawsze jest możliwe zamrożenie jego otoczenia oraz niektórych scalonych kontrolerów karty procesora. Traci się w ten sposób warunki pracy w czasie rzeczywistym, ale dla niektórych prac jest to rozwiązanie zadowalające. Użycie tej metody nie stwarza problemów w kwestii nadążania z wizualizacją danych, gdyż praca oprogramowania prototypu jest wstrzymana.

b) DOSTĘP DO PŁATU PAMIĘCI — Dostęp do danych poprzez pamięć emulowaną dwubramową. W czasie odczytu zmienionych danych emulator układowy lub pamięci wymaga na krótki okres wyłączności dostępu (forsowany sygnał WAIT lub szybka pamięć z przeplotem czasowym). Problemem przy użyciu tej metody może być kwestia ich wizualizacji (czy oprogramowanie emulatora nadąży) oraz kwestia

ich aktualności (czy śledzony obszar danych nie zmieni się powtórnie). Próba zamrażania pamięci na czas jej odczytu i wizualizacji może jednak spowodować utratę warunków pracy w czasie rzeczywistym.

c) DOSTĘP DO KOLEJKI ZMODYFIKOWANYCH DANYCH — Zapis modyfikowanych danych do bufora wprost z magistrali, w cyklach zapisu do pamięci. Rozwiązanie to wymaga odpowiednio dużej pojemności pamięci układu, który będzie dane odczytywał i kolejkował. Rozwiązanie to grozi utratą ciągłości informacji o zmianach danych, gdy oprogramowanie emulatora nie nadąży z opróżnianiem bufora odczytanych wartości lub gdy oprogramowanie emulatora nie nadąży z ich wizualizacją, natomiast nie grozi utratą warunków pracy w czasie rzeczywistym.

Niezależnie od konkretnej sprzętowo-programowej realizacji dostępu do danych emulowanego prototypu załóżmy, że mamy zdefiniowaną procedurę dostępu do danych o następującym nagłówku:

```
function GetMem (Address : integer) : Byte;
```

pozwalającą po wykryciu zmiany danych na odczyt nowych danych z kolejnych bajtów pamięci operacyjnej. Założono, że problem utraty warunków pracy w czasie rzeczywistym jest rozwiązany na poziomie realizacji poszczególnych procedur lub przez operatora, przygotowującego eksperyment. W celu uogólnienia metody odczytu można założyć, że dla jak najmniejszej ingerencji w pracę prototypu funkcja `GetMem` odczytuje tylko zmienione wartości, natomiast wartości nie zmienione są odczytywane w podręcznej pamięci buforowej, która zawiera interesujący nas obszar. Został on przepisany przed rozpoczęciem śledzenia danych w czasie rzeczywistym do bufora procedurą o następującym nagłówku:

```
procedure Buffer (BAddress, EAddress : integer);
```

gdzie parametry `BAddress` i `EAddress` wskazują kolejno na początek i koniec buforowanego obszaru.

4. Sposób opisu śledzenia danych

Posługując się zadeklarowanymi, wirtualnymi funkcjami można zaproponować przykładowy język automatyzujący funkcje śledzenia stanu uruchamianego oprogramowania w środowisku danego systemu operacyjnego czasu rzeczywistego. Ogólna zasada polega

na rozszerzeniu konstrukcji napotkanej już w niektórych programach uruchamiających (*debuggerach*) — tzw. *WatchPoint* (na przykład debugger *CodeView*, debuggery oprogramowania Borlanda lub RTDS/2- PC). Zakładając, że jest dostępna tablica adresów i nazw symbolicznych uruchamianego oprogramowania, a zatem nazwy użyte poniżej będą zgodne z nazwami użytymi w kodzie źródłowym uruchamianego oprogramowania, można zaproponować podstawowe moduły języka deklaracji śladowania danych. Aby uprościć jego czytelność, podstawowa składnia będzie zgodna z językiem *Pascal*, zdefiniowanym jak w pozycji [4], natomiast elementy zorientowane na problem zostaną opisane poniżej.

Załóżmy, że dla celów śladowania zmian utworzymy kilka modułów definiujących kolejno stałe dla śladowania, dane do śledzenia i dane do wyświetlania. Załóżmy również, że po uruchomieniu zadeklarowanych modułów śledzenie zostanie zainicjowane, a następnie program w danym języku będzie kolejno oczekiwał na zmianę danych i wyświetlał, oczekiwał — wyświetlał aż do czasu, gdy użytkownik przerwie jego pracę. Wtedy definicja poszczególnych modułów może mieć poniższą postać.

1. Moduł definicji (DEF — ENDDEF).

   Moduł ten zaczyna się od słowa kluczowego `DEF` i kończy słowem kluczowym `ENDDEF`.

   Powinien on zawierać deklarację pamięci, w której rezydują dane uruchamianego oprogramowania, o postaci

   ```
   MEMORY = (<adres-poczatkowy>, <adres-koncowy>)
   ```

   przykładowo

   ```
   MEMORY = ($200, $1FFF)
   ```

   co deklaruje adresy fizyczne używanego obszaru lub

   ```
   MEMORY = (_Data, _EData)
   ```

   definiujące obszar wg symboli używanych przez kod źródłowy lub kompilator uruchamianego oprogramowania. Deklaracja ta może służyć do deklaracji sprzętowego buforowania danych funkcją `Buffer`. Następnym elementem opisywanego modułu powinna być deklaracja struktur danych, użytych w oprogramowaniu. Przykładowo:

```
type QUEUE = record      (* Kolejka                 *)
     Start : integer;    (* Adres poczatku kolejki *)
     End   : integer;    (* Adres konca kolejki    *)
end;

type QUEUE_ITEM = record      (* Element kolejki          *)
          Previous : integer; (* Adres poprzedniego el. *)
          Next     : integer; (* Adres nastepnego el.   *)
          Buffer   : integer; (* Adres bufora danych    *)
     end
```

Należy zauważyć, że w powyższych przykładach założono, że dana zawierająca adres ma taką samą długość, jak dana zawierająca liczbę całkowitą. Założenie to nie zawsze jest prawdziwe i wtedy należałoby utworzyć nowe typy danych lub wykorzystać znane w Pascalu typy wskaźnikowe. Tutaj jednak, dla uproszczenia zapisu, założenie to jest do przyjęcia.

Ostatnim elementem modułu deklaracji powinna być deklaracja samych śledzonych danych. Przykładowo:

```
var  FreeMessages : QUEUE;     (* Kolejki komunikatow *)
     Messages     : QUEUE;

                                (* Pula komunikatow    *)
     Message      : array [0..100] of QUEUE_ITEM;

     CurrProcess  : integer    (* Biezacy proces       *)
```

Wykorzystanie funkcji Buffer mogłoby być realizowane na podstawie powyższych deklaracji danych, które podlegają obserwacji. Wybór sposobu użycia tej funkcji zależałby oczywiście od konkretnej implementacji języka, natomiast powyżej została wprowadzona jawna deklaracja buforowanej pamięci, aby uwypuklić ten problem.

2. Definicja śledzonych danych (DATA — ENDDATA). Może ona mieć następującą przykładową postać:

```
DATA

    Message[0].Previous

ENDDATA
```

3. Definicja wyświetlanych danych oraz sposobu ich wyświetlenia (DISP — ENDDISP).

Przy realizacji tego modułu załóżmy, że dla potrzeb języka *Pascal*, na który potem dany język opisu śledzonych danych będzie tłumaczony, zrealizowano procedurę o następującym nagłówku:

```
procedure WaitAccess;     (* Oczekiwanie na zmiany sledzonych
                             danych
                          *)
```

oraz że w deklarowanym języku istnieje standardowa procedura ADDR(), dostarczająca wartość adresu danej. Należy założyć również istnienie standardowej procedury FINISH(), przerywającej pracę programu śledzenia danych.

Przykładowa deklaracja modułu może mieć następującą postać:

```
DISP

tmp : integer;

    if (Message[0].Previous = 0) then
       begin
         if (FreeMessages.Start = ADDR(Message[0]) ) then
            begin
              write('Komunikat w kolejce WOLNYCH');
              writeln(', proces = ', CurrProcess);
            end
         else
         if (Messages.Start = ADDR(Message[0]) ) then
            begin
              write('Komunikat w kolejce ZAJ/ETYCH');
```

```
                writeln(', proces = ', CurrProcess);
              end
            else
              begin
                write('Komunikat w /zadnej kolejce');
                writeln(', proces = ', CurrProcess);
                FINISH();
              end
          end
        else
          begin
              tmp := ADDR(Message[0]);
            repeat
              tmp := tmp.Previous
            until tmp.Previous = 0;
            if (FreeMessages.Start = ADDR(tmp) ) then
              begin
                write('Komunikat w kolejce WOLNYCH');
                writeln(', proces = ', CurrProcess);
              end
            else
            if (Messages.Start = ADDR(tmp) ) then
              begin
                write('Komunikat w kolejce ZAJ/ETYCH');
                writeln(', proces = ', CurrProcess);
                FINISH();
              end
            else
              begin
                write('Komunikat w /zadnej kolejce');
                writeln(', proces = ', CurrProcess);
              end
          end

ENDDISP
```

Analizując powyższą przykładową definicję języka dostrzegamy, że dla pełnej realizacji jego funkcji konieczna jest implementacja kilku podstawowych cech w oprogramowaniu emulatora. Są to kolejno:

1. Możliwość tworzenia programów lub makroinstrukcji składających się z wielu zleceń emulatora. Powinno być możliwe przechowywanie ich treści w plikach oraz parametryzowanie ich wywołań w celu tworzenia procedur. Ogólnie biorąc powinno być możliwe wydzielenie języka zleceń danego emulatora.

2. Możliwość deklaracji zmiennych lokalnych języka zleceń emulatora.

3. Istnienie w danym języku zleceń emulatora strukturalnych instrukcji warunkowej i pętli. Minimalna wersja niestrukturalna winna zawierać co najmniej instrukcję warunku, instrukcję skoku i możliwość deklaracji etykiety.

4. Możliwość detekcji dostępu do danych w czasie rzeczywistym, co jest z reguły funkcją wbudowaną w większość współczesnych konstrukcji emulatorów.

5. Możliwość odczytu danych bez utraty trybu pracy w czasie rzeczywistym. Powyższa cecha emulatora z reguły wymaga realizacji określonych modułów sprzętowych. Jako cecha konstrukcyjna jest istotna tylko w pewnych sytuacjach i nie zawsze posiada swą sprzętową realizację. Problemy sprzętowe i programowe związane z jej realizacją zostały już opisane w rozdziale 3.

5. Przykład: Analiza kolejki komunikatów

Załóżmy, że mamy następujący problem. W naszym oprogramowaniu używamy dwóch kolejek komunikatów. W jednej przechowywane są wolne bufory (FreeMessages), w drugiej informacje, np. dane z pomiarów (Messages). Stwierdzono, że na skutek błędnego działania oprogramowania gubione są bufory komunikatów. Organizacja kolejek komunikatów jest taka, że struktura kolejki wskazuje na pierwszy i ostatni element, natomiast element kolejki wskazuje na poprzedni i następny w kolejce. Jeśli element jest ostatni lub pierwszy, to odpowiedni wskaźnik (ten który nie ma na co wskazywać) jest równy 0.

Aby odnaleźć miejsce odpowiedzialne za ten błąd, wybierzemy dowolny bufor komunikatu, np. o numerze "0" i spróbujemy prześledzić jego los. Użyte zostaną do tego przykładowe definicje modułów proponowanego języka. Zestawione moduły będą miały poniższą, przykładową postać:

```
DEF
    MEMORY = ($200, $1FFF)

    type QUEUE = record      (* Kolejka                  *)
         Start : integer;    (* Adres poczatku kolejki *)
         End   : integer;    (* Adres konca kolejki    *)
    end;

    type QUEUE_ITEM = record      (* Element kolejki        *)
              Previous : integer; (* Adres poprzedniego el. *)
              Next     : integer; (* Adres nastepnego el.   *)
              Buffer   : integer; (* Adres bufora danych    *)
         end;

    var  FreeMessages : QUEUE;    (* Kolejki komunikatow *)
         Messages     : QUEUE;

                                  (* Pula komunikatow    *)
         Message      : array [0..100] of QUEUE_ITEM;

         CurrProcess  : integer;  (* Biezacy proces      *)

ENDDEF

DATA

    Message[0].Previous

ENDDATA

DISP

    tmp : integer;

    if (Message[0].Previous = 0) then
       begin
```

```
      if (FreeMessages.Start = ADDR(Message[0]) ) then
         begin
           write('Komunikat w kolejce WOLNYCH');
           writeln(', proces = ', CurrProcess);
         end
      else
      if (Messages.Start = ADDR(Message[0]) ) then
         begin
           write('Komunikat w kolejce ZAJETYCH');
           writeln(', proces = ', CurrProcess);
         end
      else
         begin
           write('Komunikat w zadnej kolejce');
           writeln(', proces = ', CurrProcess);
           FINISH();
         end
   end
else
   begin
         tmp := ADDR(Message[0]);
     repeat
        tmp := tmp.Previous
     until tmp.Previous = 0;
     if (FreeMessages.Start = ADDR(tmp) ) then
        begin
          write('Komunikat w kolejce WOLNYCH');
          writeln(', proces = ', CurrProcess);
        end
     else
     if (Messages.Start = ADDR(tmp) ) then
        begin
          write('Komunikat w kolejce ZAJETYCH');
          writeln(', proces = ', CurrProcess);
        end
     else
```

```
            begin
              write('Komunikat w zadnej kolejce');
              writeln(', proces = ', CurrProceśs);
              FINISH();
            end
        end

ENDDISP
```

Program wynikowy w języku *Pascal*, wygenerowany na podstawie tablicy adresów oraz powyższy tekst programu, powinien mieć następującą postać:

```
Program TraceData;

const                          (*  Modul DEF *)

     FreeMessage =    ;  (* Wartosci nie podane sa znane z
                             tablicy symboli programu - sa  to
                             adresy danych uruchamianego prog-
                             ramu
                          *)
     Messages     =    ;
     Message      =    ;
     CurrProcess  =    ;
     Start        = 0;     (* Pola rekordow - odleglosc od  po-
                              czatku                          *)
     End          = 4;     (* Dlugosc adresu - zalozone 4 bajty *)
     Previous     = 0;
     Next         = 4;
     Message0     = Message + 0 * 12;   (* Pozycja 0 w tablicy *)


var                            (* Dane robocze *)
     Finish, tmp : integer;

function GetInt(Addr : integer) : integer;
```

```
(*   Procedura pobiera z zadanego adresu dana 32-bitowa, ktora
 *   jest adresem innej danej lub liczba calkowita.
 *
 *   Zaklada sie organizacje danych jak INTELa.
 *)

begin
     GetInt := GetMem(Addr) + GetMem(Addr+1)*256 +
               GetMem(Addr+2)*256*256+
               GetMem(Addr+3)*256*256*256
end;

begin

     Buffer($200, $1FFF);      (* zlecenie MEMORY *)

(* Modul DATA *)

     DetectData('$ADDRESS = Message0 + Previous');

     PutCPU;          (* Start emulowanego prototypu *)

(* Modul DISP *)

     Finish = 0;
     while (Finish = 0) do
        begin
          WaitAccess;
     if (GetInt(Message0+Previous) = 0) then
       begin
          if (GetInt(FreeMessages+Start) = Message0) then
             begin
               write('Komunikat w kolejce WOLNYCH');
               writeln(', proces = ', GetInt(CurrProcess));
             end
          else
```

```
      if (GetInt(Messages+Start) = Message0) then
         begin
           write('Komunikat w kolejce ZAJETYCH');
           writeln(', proces = ', GetInt(CurrProcess));
         end
      else
         begin
           write('Komunikat w zadnej kolejce');
           writeln(', proces = ', GetInt(CurrProcess));
                Finish = 1;
         end
   end
else
   begin
         tmp := Message0;
     repeat
        tmp := GetInt(tmp+Previous)
     until GetInt(tmp+Previous) = 0;
     if (GetInt(FreeMessages+Start) = tmp) then
        begin
          write('Komunikat w kolejce WOLNYCH');
          writeln(', proces = ', GetInt(CurrProcess));
        end
     else
     if (GetInt(Messages+Start) = tmp) then
        begin
          write('Komunikat w kolejce ZAJETYCH');
          writeln(', proces = ', GetInt(CurrProcess));
        end
     else
        begin
          write('Komunikat w zadnej kolejce');
          writeln(', proces = ', GetInt(CurrProcess));
               Finish = 1;
        end
   end
```

```
        end

end.
```

Dla konkretnego emulatora (tutaj RTDS/2–PC) po odczytaniu tablicy symboli, plik realizujący powyższy algorytm (w ramach konstrukcyjnych możliwości emulatora) będzie miał następującą postać.

```
*
*    Modul DEF
*
XN End = 4
XN Next = 4
XN Message0 = Message
XN Message0 = Message0 + 0*12
*
*  Buforowanie w RTDS2-PC jest automatyczne, ale nie
*  dostosowane do optymalizacji dostepu do danych.
*

*
*    Modul DATA
*
2C wrm AT= Message0 any
BR C2 A D

*
*    Modul DISP
*
:DISP

G
:petla_G
*
*  Opisywany emulator nie posiada modulow sprzetowo-
*  programowych, pozwalajacych na dostep do danych w sposob
*  inny niz przez kanal komunikacyjny emulatora.
```

```
*
 .if $GO .goto petla_G

*
*  Dla uproszczenia zapisu algorytmu zaklada sie dostep
*  automatyczny do slow 32-bitowych slowem kluczowym 'dword'
*  oraz dane 32-bitowe identyczne z adresem.
*

 .if (dword Message0) .goto IF1
     .if !(dword FreeMessages == Message0) .goto IF01
          df CurrProcess Komunikat w kolejce WOLNYCH, proces = \%d
          .goto ENDIF
:IF01
     .if !(dword Messages == Message0) .goto IF02
          df CurrProcess Komunikat w kolejce ZAJETYCH , proces = \%d
          .goto ENDIF
:IF02
     df CurrProcess Komunikat w zadnej kolejce, proces = \%d
     .goto FINISH
:IF1
     XN tmp = Message0
:REPEAT
     XN tmp1 = tmp
     XN tmp1 = tmp1 + Previous
     XN tmp  = dword tmp1
:UNTIL
     XN tmp1 = tmp
     XN tmp1 = tmp1 + Previous
     XN tmp1  = dword tmp1
     .if !tmp1 .goto REPEAT

     .if !(dword FreeMessages == Message0) .goto IF11
          df CurrProcess Komunikat w kolejce WOLNYCH, proces = \%d
          .goto ENDIF
:IF11
```

```
    .if !(dword Messages == Message0) .goto IF12
        df CurrProcess Komunikat w kolejce ZAJETYCH , proces = \%d
        .goto ENDIF
:IF12
    df CurrProcess Komunikat w zadnej kolejce, proces = \%d
    .goto FINISH
:ENDIF

 .goto DISP

:FINISH
```

6. Wnioski

Użycie ogólnej, publikacyjnej wersji języka *Pascal* i jej automatyczna translacja na specjalizowany, skonstruowany dla innych potrzeb, język jest pożądana; zapis tego algorytmu od razu w języku emulatora, zorientowanym na ergonomiczność pracy z konsoli, jest skomplikowany.

Analiza powyższego przykładu nasuwa wniosek, że śledzenie danych oprogramowania jest silnie zależne od kompilatora, w którym oprogramowanie to zostało napisane. Dotyczy to zarówno sposobu generacji adresów, rozmiaru danych, jak i notacji ich nazw. Idealnym rozwiązaniem jest użytkowanie własnych kompilatorów, np. INTEL oferuje emulatory swych procesorów wraz z systemem operacyjnym i kompilatorami. Ogólnie biorąc, śledzenie struktur danych generowanych przez kompilator języka programowania wymaga zasad ich organizacji na poziomie fizycznym i kodu źródłowego.

Niezależnie od przyjętego założenia o dostępie do danych bez wstrzymywania pracy emulowanego prototypu, rozwiązania zapewniające postulowaną ciągłość pracy w czasie rzeczywistym są rzadko spotykane. Wybierając określony typ emulatora dla implementacji na nim proponowanego języka opisu, śledzenia i wizualizacji danych, należałoby zrezygnować z ciągłej pracy w czasie rzeczywistym albo z powodu braku odpowiednich modułów sprzętowych, albo ze względu na zagrożenie utraty danych.

Przedstawiona powyżej propozycja określonego użytkowania emulatorów układowych do uruchamiania systemów czasu rzeczywistego jest, jak już powiedziano, logicznym przedłużeniem znanej w wielu debuggerach obserwacji danych deklarowanych jako WatchPoint (punkt obserwacji). Istnieją również procesory np. 80386 i 80486, ma-

jące już tę możliwość wbudowaną w ich architekturę. Sprzętowo-programowa jej realizacja przy użyciu emulatora układowego lub emulatora pamięci pozwala uruchamiać oprogramowanie w warunkach czasu rzeczywistego, jednak złożoność danych skłania do definicji określonego języka, który pozwoliłby decydować o reakcji na określone stany obserwowanego systemu. Napotykane współcześnie konstrukcje emulatorów z reguły nie pozwalają na pełną implementację tego języka bez utraty trybu pracy w czasie rzeczywistym emulowanego prototypu, chociaż komponenty tych konstrukcji w pełni to umożliwiają. Należy oczekiwać, że dalszy rozwój oprogramowania sterowników mikroprocesorowych w językach wyższego poziomu spowoduje rozwój kolejnych konfiguracji sprzętowo-programowych emulatorów, przystających w większym stopniu do postulowanych w powyższym opracowaniu wymogów.

LITERATURA

[1] Intel — "ICE-86A/ICE-88A MICROSYSTEM IN-CIRCUIT EMULATOR OPERATING INSTRUCTION FOR ISIS-II USERS", Intel 1982.

[2] Microtec — "MICE–II Users Guide", Microtec 1986.

[3] Intel — "ICE–196 PC User Guide", Intel 1987.

[4] Dariusz Zonenberg — "Emulator układowy mikroprocesorów jako narzędzie uruchomieniowe w ujęciu programistycznym.", Wydawnictwo SIGMA-NOT, mies. INFORMATYKA nr 2, Warszawa 1992.

[5] MERA-Elzab — "RTDS/2-PC. Dokumentacja użytkowa.", MERA-Elzab Zabrze 1989.

[6] Microsoft Corp. — "Microsoft CodeView — Window Oriented Debugger for the MS-DOS Operating System", Microsoft 1987.

[7] Per Brinch Hansen — "Podstawy systemów operacyjnych", Wydawnictwa Naukowo-Techniczne, Warszawa 1979.

Recenzent: Prof. dr hab. inż. Andrzej Grzywak

Wpłynęło do Redakcji 28 listopada 1991 r.

Abstract

The article presented real time microprocessor system software as changes in data structures performed during alghoritm execution. Next, the possibilities were described of data tracing in a real time environment that can be achieved using known features of contemporary ICEs. After a preliminary problem presentation, an example description language was presented for data tracing and presentation of microprocessor systems debugged using an ICE. The language was *Pascal*-like because of *Pascal*'s readability and popularity. A simple debugging task was discussed to illustrate the language's application. The example language was used to describe the debugging task and afterwards it was translated into a real RTDS/2–PC ICE language. Conclusions points to close connections between the tasks of software debugging and programming language compilers as a directive for ICE software development. ICE design features necessary for the presented application area were also indicated.