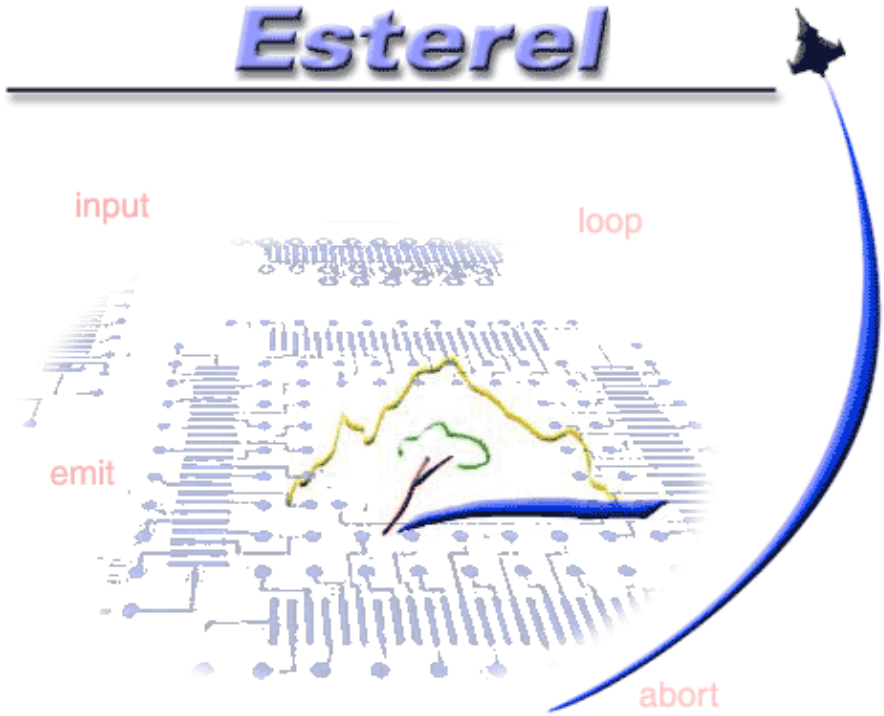
Zilog 2003 Contest Entry Z4303

Z8 Encore! Esterel proof of concept

The Esterel Language

Esterel: a Synchronous Reactive Programming Language

The language with the built in RTOS.



"Correct−by−construction design is the only practical solution to the problems strangling the productivity of embedded software and electronic systems developers."
−− Eric Bantegnie CEO of Esterel Technologies.

Table of Contents

Abstract

Since the language Esterel is being used to create mission−critical embedded software systems like AIRBUS A340 fly−by−wire, subway signaling systems, and nuclear power power plants, I wanted to find out if I could use Esterel on a small micro like the Z8 Encore!.

The hardware for this project is the stock Zilog ZDS−II evaluation board, with the addition of four 1k 1/8W resistors. As a proof of concept that Esterel can be run on the Z8 Encore! line of parts. I have ported Gérard Berry's et.al., Esterel "Reflex Game" to run on the Zilog ZDS−II evaluation board.

This demonstrates that "Correct−by−construction design" methodologies are possible on small embedded systems such as the Z8 Encore!.

While the program here is presented as a game, it does have practical commercial application in a repackaged form. It can be used to see if you are to sleep deprived from overtime to be working on software; tired programmers make mistakes, or to impaired to drive after consumption of alcohol.

Definitions

- Concurrently: Apparently at the same time.
- Reactive system: A system that is in continuous interaction with its environment. A output from the system may be feed back to the system input via the environment.
- Requirement: A requirement can be any need or expectation for a system or for its software. Requirements reflect the stated or implied needs of the customer.
- Simultaneously: At the same time.
- Specification: A specification "means any requirement with which a product, process, service, or other activity must conform." (See 21 CFR§820.3(y).)
- Validation: Have we built the correct device? Do we meet the customer's requirements?
- Verification: Have we built the device correctly? Did we find and remove all of the 'bugs'?

> Requirements are a statement of what the customer wants and needs. *Requirements* are used for *validation*. Specifications are the documentation of how the customer requirements are met by the system design. *Specifications* are used for *verification*.

Introduction

Esterel is both a programming language, dedicated to programming reactive systems, and a compiler that translates concurrent Esterel programs into single−threaded C or Verilog programs. Because of Esterel's compositional facilities, you can use it to write compact specifications for complex embedded systems.

Esterel is also well suited to developing protocol state machines, typical of Internet protocols. A example HDLC protocol may be found in the references.

The goal of Esterel is unambiguous specifications that can generate an automated implementation guaranteed to match the specification 100%. It comes down to what we *specify* should correspond exactly to what we finally *execute*.

Consider the following controller specification written in natural language:

> *Emit the output* O *as soon as both the inputs* A *and* B *have been received.*
> *Reset the behavior whenever the input* R *is received.*

In Esterel, the code is written as follows:

```
module ABRO:
input A, B, R;
output O;
loop
 [ await A || await B ];
 emit O
each R
end module
```

Since the generated code is guaranteed to match the specifications, separate verification steps typical of Spin/Promela modeling can be done away with freeing valuable time and resources.

However this does not mean that there can not be mistakes even in Esterel code, as I show in the Reflex Game Esterel source code example. There is not yet a panacea that overcomes human error in writing specifications.

Synchronous Time Model

Esterel is one of a family of synchronous languages, like SyncCharts, Lustre, Argos or Signal, which are particularly well−suited to programming reactive systems, including real−time systems and control automata.

In the Synchronous Time Model reactions are presumed to be instantaneous. This represents the assumption that the microprocessor doing the calculations, have response times orders of magnitudes faster than the events of the real world environment. The output events emitted by a reaction to an input event are considered to occurs synchronously with the input event.

In this view of time, time is measured in *Instances*. Activity takes place only during a active *Instance*. Between *Instances* the Esterel code is idle.

This counterintuitive view of time allows us to no longer worry about such things as simultaneous access to variables by different tasks, semaphores, and critical sections in Esterel code, which are typical of details we must code when using Real Time Operating Systems.

The Esterel compiler schedules such tedious details for us, removing the burden from us, giving a savings in the time by not writing the now unnecessary low level code, and improvement in reliability because of some overlooked multiple thread access to a resource that we may have overlooked.

Esterel In A Nutshell

The Esterel language can be thought of as a language for transforming a specification document in to concurrent nested state machines.

Esterel is an imperative concurrent language whose model of time resembles that in a synchronous digital logic circuit. The execution of the program progresses a cycle at a time and in each cycle, the program computes its output and next state based on its input and the previous state. In each cycle, the program performs a bounded amount of work; no intra−cycle loops are allowed.

Esterel is a concurrent language in that its programs may contain multiple threads of control. Unlike typical multi−threaded software systems, however, Esterel's threads execute in lockstep: each sees the same cycle boundaries and communicates with other threads using a disciplined broadcast mechanism.

Esterel's threads communicate through signals, which behave like wires in digital logic circuits. In each cycle, each signal takes a single Boolean value ("*present*" or "*absent*") that does not automatically persist between cycles. Esterel's communication run is simple: within a cycle, any thread that reads the value of a signal must wait for any other threads that set the value of a signal.

Statements in Esterel either execute within a cycle (e.g.,*emit* makes a given signal present in the current cycle, *present* tests a signal's presence) or take one or more cycles to complete (e.g., *pause* delays a cycle before continuing, *await* waits for a cycle in which a particular signal is present). Preemption statements check a condition in every cycle before deciding whether to allow their bodies to execute. For example, the *every* statement performs a reset−like action by restarting its body in any cycle in which its predicate is true.

"Esterel implemented in software simulates simultaneity with concurrency."
−− Professor Stephen A. Edwards Columbia University

Esterel Hardware Interfacing

While the Esterel compiler will produce different output formats, I'll only described the C interface as it applies to the Z8 Encore!.

The Esterel/C interface is done via a '#include "program_name.h"' directive in the Esterel compiler generated code. It must contain the interface to types, constants, functions, and procedures.

Each procedure used in the Esterel program must have a C definition, using either a #define directive in program_name.h or a standard C function definition in some other file. If not #defined, the C function is automatically declared to be extern in program_name.c.

Input Signals

For each input signal *IS*, the Esterel compiler generates an *input C function* called PROGRAM_NAME_I_IS, which takes an argument of the appropriate type.

If we are compiling a program called *DISPLAY*, this Esterel declaration

```
input WATCH_MODE_COMMAND;
```

will expect a #define or prototype in DISPLAY.H that ultimately will be resolved by a C file containing:

```
void DISPLAY_I_WATCH_MODE_COMMAND () {...}
```

When a program *PROG* should react to an input event composed of one or more simultaneous input signals, the associated input C function(s) should be called before calling the main Esterel execution function *PROG*.

If several input functions are called before calling *PROG*, the corresponding input signals are considered as forming the current input event of the reaction. The input signals are considered as being simultaneous. Therefore, the notion of "simultaneous signals" is a purely logical one at the C level. Two signals are considered as simultaneous at that level as long as their input C functions are both called before calling the reaction function. Which signals are to be considered as simultaneous and when to call the Esterel automaton is entirely left to the user.

> ***WARNING**: Reactions are not reentrant and must be executed in an atomic way. During the execution of the reaction function, neither user input C functions nor the reaction function itself can be called.*
>
> > This means that all inputs to the Esterel reaction code must be stable, and not changed by any interrupt. I show how to deal with this in examples that follow.

Output Signals

For each output signal *OS*, the user must write a void output C function *PROG_O_OS* that

takes an argument of the appropriate type.

***WARNING**: The order of the output function calls performed by the reaction function is arbitrary and unspecified.*

Assume that a reaction causes the output of a pure signal OS1 and of an integer signal OS2 with value 63. Then DISPLAY calls the user−defined C functions DISPLAY_O_OS1 and DISPLAY_O_OS2 with the appropriate arguments; the following calls will be executed (in arbitrary order) in the body of the Esterel DISPLAY code:

```
DISPLAY_O_OS1();
DISPLAY_O_OS2(63);
```

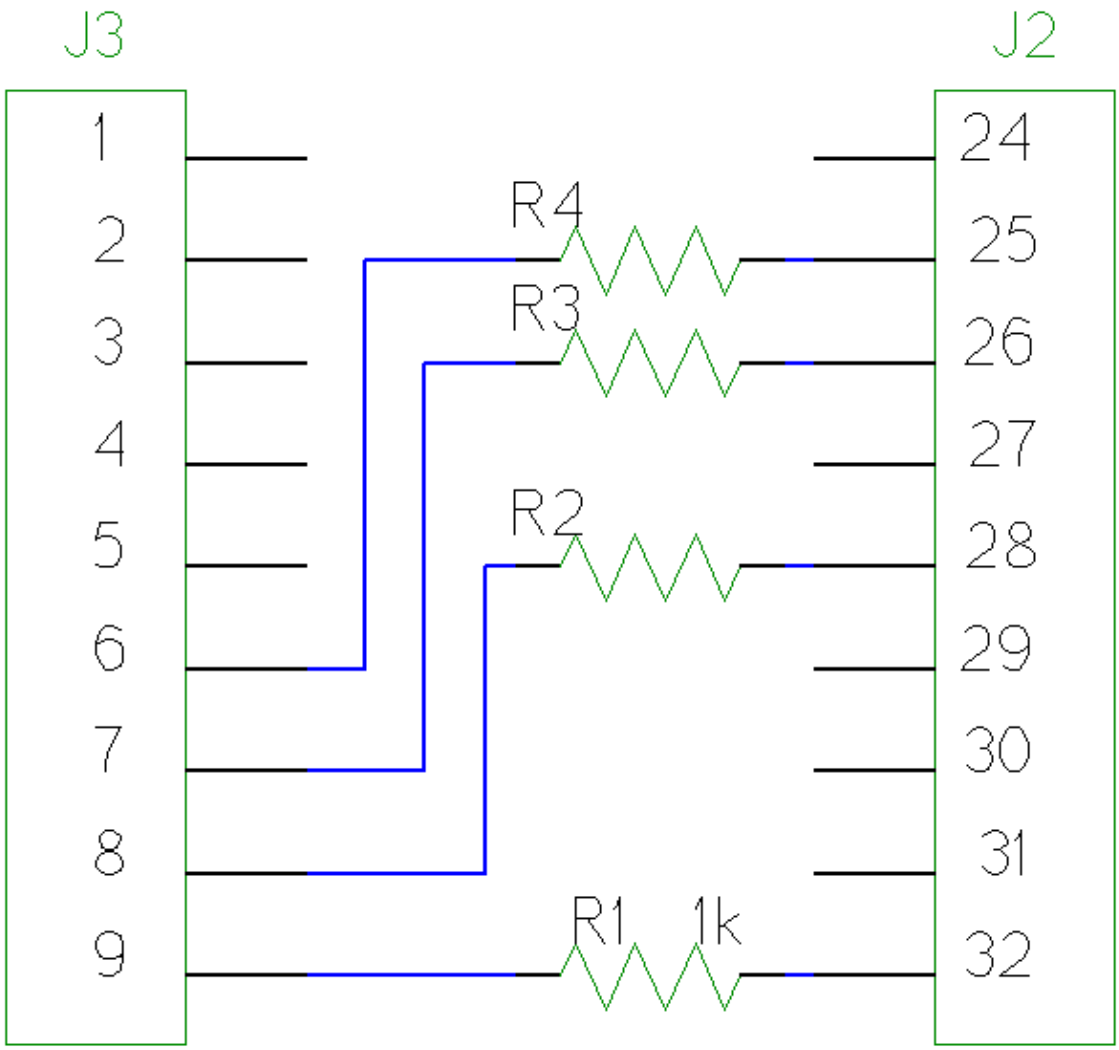
The C functions DISPLAY_O_OS1 and DISPLAY_O_OS2 must do whatever is necessary to communicate with the actual physical hardware environment.

Conclusion

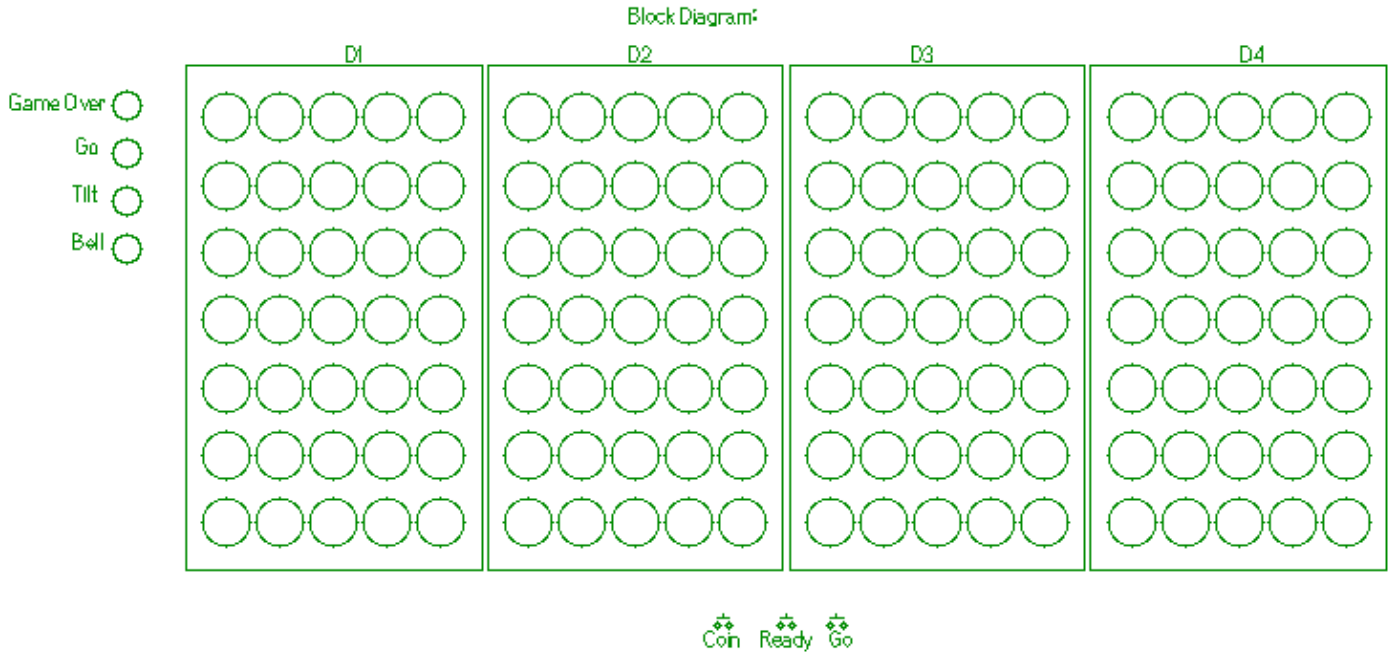
I found porting Esterel to the Z8 Encore! to be easy, and now I can use my ZDS−II to test my reflexes, when I'm not developing other code with it.

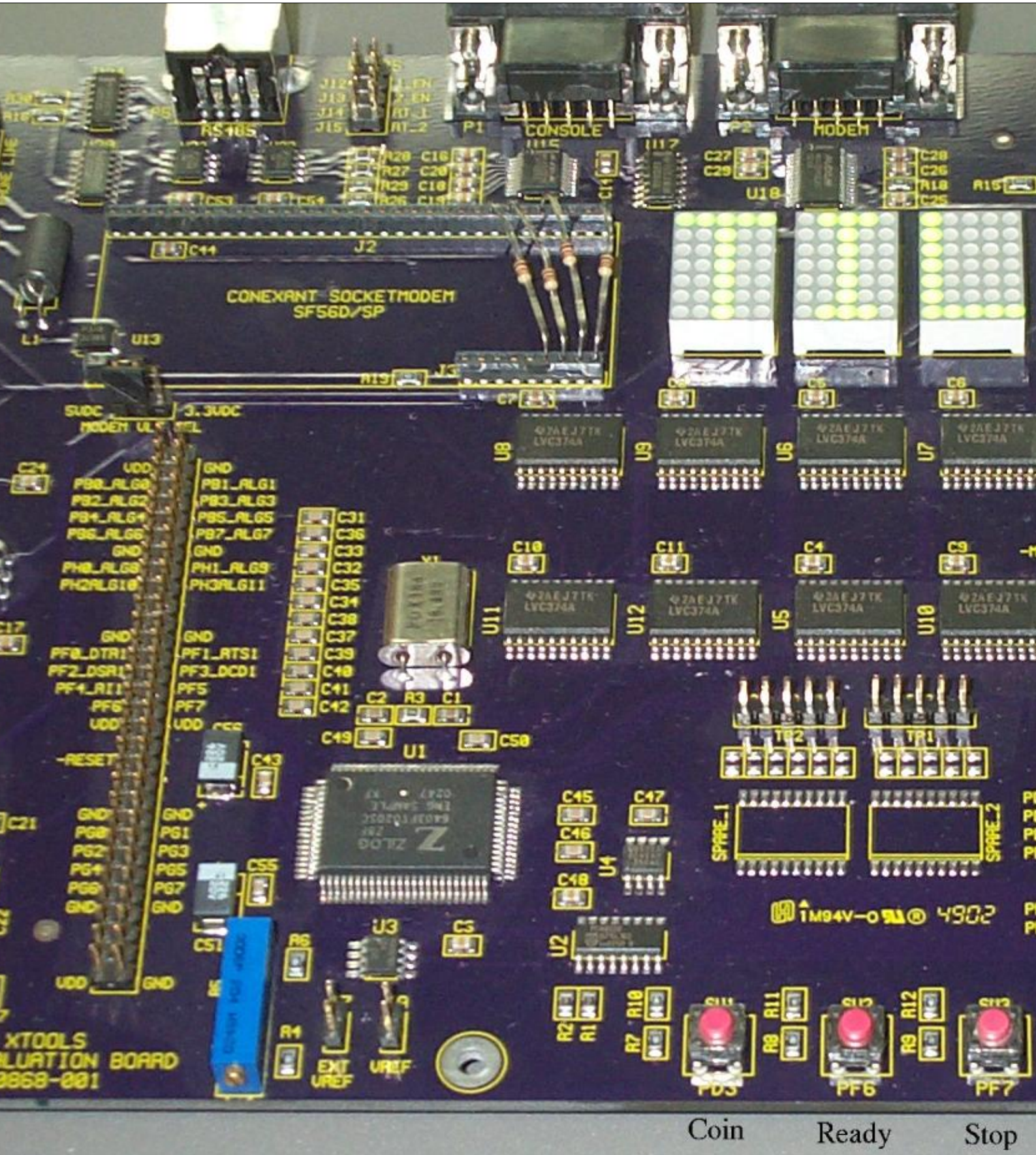
Schematic, Photo and Block Diagram

Hardware Block Diagram
and
Schematics of ZDS−II Evaluation board



The only required modification to the stock ZDS−II is the addition of four 1k 1/8 Watt resistors.


Block Diagram:
D1
D2
D3
D4
Game Over
Go
Tilt
Bell
Coin
Ready
Go


RS485
P1
CONSOLE
P2
MODEM
J2
CONEXANT SOCKETMODEM
SF56D/SP
J3
U13
U18
U8
U9
U6
U7
U11
U12
U5
U10
U1
U4
U2
U3
TP2
TP1
SPARE_1
SPARE_2
XTOOLS
EVALUATION BOARD
EXT VREF
VREF
PD3
PF6
PF7
Coin
Ready
Stop

Thanks To

Gérard Berry at Esterel Technologies to use his Esterel reflex game code, and for permission to quote from other examples in the distribution and literature. Esterel Technologies graciously makes available the text based Version−5.92 compiler as a free download from their web site. It runs under Linux, Solaris and Windows. They sell enhanced graphical design suites for Esterel development.

Prof. Stephen A. Edwards at Columbia University for assistance in getting up to speed with Esterel, and permission to quote his *Esterel in a Nutshell*, from a unpublished paper.

The Columbia Esterel Compiler is an open−source compiler designed for research in both hardware and software generation from the Esterel synchronous language. It currently supports a subset of Esterel V5, and can generate either C, Verilog or BLIF circuit descriptions from an Esterel program. It is licensed under the BSD license.

References

Gérard Berry, architect of the Esterel language, considers this to be the seminal reference on Esterel:

> *The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation*, G. Berry and G. Gonthier. *Science of Computer Programming*, vol. 19, n.2 (1992) 87−152.

The Esterel v5 Primer, G. Berry.

Incremental development of an HDLC entity in Esterel, G. Berry and G. Gonthier. *Computer Networks and ISDN Systems* 22, (1991) 35−49.

An Introduction to Esterel by Girish Keshav Palshikar in *Embedded Systems Programming* magazine; Nov/2001.

The Synchronous Language Esterel by Prof. Stephen A. Edwards.

Programming in Esterel by Prof. Stephen A. Edwards.

The Synchronous Languages 12 Years Later, Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. Proceedings of the IEEE 91(1):64−83, January 2003.

Esterel Examples

Example1

Esterel Code

Example One Lights LED1 after SW1 (PF7) is pressed followed by the pressing of SW2 (PF6).

```
% Single line comments start with "%".  Multi−line comments are enclosed
% within tags of "%{}%".

% Example1

%{
  Light LED1 after SW1 then SW2 is pressed, in that order.
}%

module Example1 :

% Instruct the Esterel compiler to generate #include "example1.h":

type ForceTheIncludeDirective;

       % Interface Declaration (objects a module imports or exports):

       input  SW1_ASSERTED;  % Switch SW1 input
       input  SW2_ASSERTED;  % Switch SW2 input

       output LED1_ASSERT;   % LED1 Output

       % Statement Body:

       await SW1_ASSERTED;   % Wait for SW1 to be pressed
       await SW2_ASSERTED;   % followed by SW2

       emit  LED1_ASSERT;    % then light LED1

end module
```

C Code

example1.c shows the resultant C output generated by the Esterel compiler.

main.c shows the C to Esterel interface.

Example2

Esterel Code

Example two blinks one of the LEDs on the ZDS−II board once per second.

```
% Single line comments start with "%".  Multi−line comments are enclosed
% within tags of "%{}%".
```

```
% Example2

%{
  Blink LED2 every second.
}%

module Example2 :

% Instruct the Esterel compiler to generate #include "example2.h":

type ForceTheIncludeDirective;

       % Interface Declaration (objects a module imports or exports).:

       input  Second_Tick;      % Time Base Tick from IRQ

       output LED2_TOGGLE;      % LED2 Output

       % Statement Body:

        every Second_Tick do
                emit  LED2_TOGGLE;     % Toggle the LED every second
        end every

end module
```

C Code

example2.c shows the resultant C output generated by the Esterel compiler.

main.c shows the C to Esterel interface.

Example3

Esterel Code

Example Three is the combination of Example One and Example Two.

```
% Single line comments start with "%".  Multi-line comments are enclosed
% within tags of "%{}%".

% Example3

%{
  Show how to combine the two independent Example1 and Example2 programs,
  into one resultant program.
}%

module Example3 :

% Instruct the Esterel compiler to generate #include "example3.h":

type ForceTheIncludeDirective;

% I/O required by Example1:
       input  SW1_ASSERTED;  % Switch SW1 input
       input  SW2_ASSERTED;  % Switch SW2 input
       output LED1_ASSERT;   % LED1 Output
```

```
% I/O required by Example2:
       input  Second_Tick;   % Time Base Tick from IRQ
       output LED2_TOGGLE;   % LED2 Output

% Run Example1 and Example2 concurrently, in endless loop:
        loop

             run Example1
         ||

             run Example2

        end loop

end module
```

C Code

example3.c shows the resultant C output generated by the Esterel compiler.

main.c shows the C to Esterel interface.

Reflex Game Specification

We want to program the following reflex game machine.

The player controls the machine with the three buttons on the ZDS−II:

- putting a coin in a COIN slot, to start the game via SW1 (PD3).
- pressing a READY button to start a reflex measure via SW2 (PF6).
- pressing a STOP button to end a measure via SW3 (PF7).

The machine reacts to these commands by operating the following devices, which use the ZDS−II MODEM LED's connected to J3:

- a numerical display DISPLAY that displays reflex times
- a GO lamp that signals the beginning of a measure via DCD LED
- a GAME_OVER lamp that signals the end of a game via RX LED
- a TILT lamp that signals that the player has tried to cheat or has abandoned the game via DTR LED
- a RING_BELL that rings when the player hits the wrong button via TX LED

When the machine is turned on the display shows the digit zero, the GAME_OVER lamp is on the GO and TILT lamps are off. The player then starts a game by inserting a coin. The normal behavior is as follows:

Each game is composed of a fixed number MEASUREUMBER of reflex measures, five in our sample program. A measure starts when the player presses the READY button; then after a random time the GO lamp turns on and the player must press the STOP button as fast as he can. When he does so the GO lamp turns of and the reflex time measured in milliseconds is displayed on the numerical display. A new measure starts when the player presses READY again. When the cycle of MEASUREUMBER measures is completed the average reflex time is displayed after a pause of PAUSE_LENGTH milliseconds and the GAME_OVER lamp is turned on.

There are five exception cases. Two of them are simple mistakes and make the bell ring:

- the player presses STOP instead of READY to start a measure
- the player presses READY during a measure

In the next three cases the TILT and GAME_OVER lamps are turned on the GO lamp is turned on and the game ends:

- when he is supposed to press the READY button the player does not press it within LIMIT_TIME milliseconds (one considers that the player has abandoned the game)
- when he is supposed to press the STOP button (that is after the GO light turns on) the player does not press it within LIMIT_TIME milliseconds (this is also considered as an abandon)
- after he has pressed the READY button, the player presses the STOP button before the machine turns the GO lamp on, or at the same time this happens (the player tried to cheat).

A last anomaly appears if the player inserts a coin during a game. Then a new game is started afresh at once.

- as there is no built−in time in Esterel, we must explicitly declare a MS signal for milliseconds. We use one of the Z8 Enhanced! timer's to produced this system 'tick'.
- we make some choices for relations between input signals: e.g., we notice that the specification does not tell what should happen if both buttons are pressed simultaneously, so we assume they never are.

Reflex Game

Esterel Code

Reflex Game Esterel Source Code.

```
% Single line comments start with "%".  Multi−line comments are enclosed
% within tags of "%{}%".

% The Reflex Game Program

%{
  To illustrate the RTOS like behavior of Esterel, we use an auxiliary
  module AVERAGE to compute the average reflex time.

  AVERAGE and REFLEX_GAME are considered to be two separate programs
  communicating via the reception and emission of signals.

  The AVERAGE module's purpose is to receive integers and to broadcast the
  average of the integers received so far. The communication with AVERAGE
  involves two signals:

  * INCREMENTAVERAGE(integer): input of the AVERAGE module; provokes the
    incrementation of the current average value by the conveyed integer

  * AVERAGE_VALUE(integer): output of the AVERAGE module; broadcasts the
    current average value.

    The new average value is emitted synchronously with any input.

    You must realize that the signal AVERAGE_VALUE is undefined up to the
    first reception of INCREMENT_AVERAGE. Reading an undefined signal is
    flagged as an error by the simulator, but not by the compiler.
```

```
  Note that you have played the game enough when the AVERAGE module
  crashes with a division by zero error.  This happens if MEASURE_NUMBER
  is set high enough (unlikely) to cause the value of a integer to wrap
  around to zero.  TOTAL is also likely to wrap around giving wrong
  results.

  We don't try to trap those here for the sake of clarity in this simple
  example.  This shows that Esterel can produce code that crashes if you
  get complacent in your requirements and specifications.  No programming
  language yet has over come the human ability to specify things that are
  wrong.

  Esterel's strength is its robust real time signal manipulation,
  not its numerical manipulation.
}%

module AVERAGE:

% Instruct the Esterel compiler to generate #include "reflex_game.h":

type ForceTheIncludeDirective;

input INCREMENT_AVERAGE : integer;

output AVERAGE_VALUE : integer;

var TOTAL := 0 : integer,
  NUMBER := 0 : integer in
  every immediate INCREMENT_AVERAGE do
    TOTAL := TOTAL + ?INCREMENT_AVERAGE;
    NUMBER := NUMBER + 1;
    emit AVERAGE_VALUE(TOTAL / NUMBER)
  end every
end var

end module

% AUTOMATON ENGINE:

module REFLEX_GAME:

% CONSTANTS:

constant LIMIT_TIME : integer;
constant MEASURE_NUMBER : integer;
constant PAUSE_LENGTH : integer;

%{
  We need to wait for a random time. To determine the delay length, we call an
  external function RANDOM.

  Notice that such a "function" is somewhat improper in Esterel, since
  functions should be deterministic. To be perfectly clean, we could send a
  signal START_TIMER to an external random timer and wait for a TIME_EXPIRED
  reply. However such a use of some random number generator is obviously
  standard practice even in deterministic languages.
}%

function RANDOM() : integer;
```

```
% INPUT FUNCTIONS:

input MS;
input COIN;
input READY;
input STOP;

% OUTPUT ACTIONS:

output DISPLAY : integer;
output GO_ON;
output GO_OFF;
output GAME_OVER_ON;
output GAME_OVER_OFF;
output TILT_ON;
output TILT_OFF;
output RING_BELL;

%{
  [Note: The 'relation' directive is a hold over from older compiler
     versions. Esterel−Technologies V5 or later compiler and all of CEC's,
     do not have such a restriction and in fact simply ignore the
     'relation' directives.
  ]

  Although it is not strictly necessary, we shall assume the following
  incompatibility relations between input signals.
}%

relation MS # COIN # READY;
relation COIN # STOP;
relation READY # STOP;

% REFLEX_GAME proper starts at this point.

% Overall initializations, such as setting up hardware at power up:

% Display zero on the display, turn GO and TILT LED off, GAME_OVER LED on:

emit DISPLAY(0);
emit GO_OFF;
emit TILT_OFF;
emit GAME_OVER_ON;

% Loop over a single game:

every COIN do

% Initializations, done every time we insert a coin for a new game:

  emit DISPLAY(0);
  emit GO_OFF;
  emit GAME_OVER_OFF;
  emit TILT_OFF;

  % Exception handling:

  trap END_GAME,
       ERROR in
     signal INCREMENT_AVERAGE : integer,
     AVERAGE_VALUE : integer in
```

```
[

%{
  We use the AVERAGE module via a 'run' instruction, putting it in
  parallel with the body of the END_GAME trap construct.

  Since AVERAGE is effectively copied inside the body of the main
  "every COIN" loop via 'run', it is restarted afresh whenever
   a new coin is inserted.
}%

run AVERAGE

% Parallelize 'Average' module and rest of game:

||

repeat MEASURE_NUMBER times

  % Phase−1: waiting for the READY button:
  %{
    During phase 1, we wait for the player to press READY, watching
    the time limit of LIMIT_TIME milliseconds and ringing the bell
    whenever STOP is pressed. If the timeout elapses, we exit the
    ERROR trap:
  }%

  do
   do
    every STOP do
    emit RING_BELL
      end every
    upto READY

  watching LIMIT_TIME MS timeout
   exit ERROR
  end timeout;

  % Phases 2 and 3:

  trap END_MEASURE in
   [

   %{
     There is something in common between phases 2 and 3: pressing
     the  READY  button rings  the  bell. This  is  conveniently
     expressed using  a parallel construct, putting the following
     statement in parallel with the phase2/phase3 sequence:
   }%

   every READY do
       emit RING_BELL
   end every

   ||

   % Phase−2: waiting for RANDOM MS

   %{
     During phase 2, we wait for a random number of milliseconds,
     raising ERROR if STOP is pressed before the end of the delay.
```

```
      In other words we wait for a random delay  watching STOP and
      exit ERROR in case of timeout:
    }%

    do
     await RANDOM() MS
      watching STOP timeout
       exit ERROR
      end timeout;

    %{
      Notice that the Phase-2 code is the "dual" of the Phase-1 code:
      the pairs READY-MS of Phase-1 and MS-STOP of Phase-2 play the
      same role. This illustrates the advantage of defining temporal
      units for all signals, and not just some predefined physical
      time clock.

      Notice also that ERROR is exited if the player presses STOP
      simultaneously with the end of the delay, according to the
      semantics of the watching statement.
    }%

    emit GO_ON;

    % Phase-3: waiting for the STOP button

    %{
      We wait for STOP with a time limit of LIMIT_TIME milliseconds,
      counting the milliseconds. The structure is similar to that of
      Phase-1, except that we must declare a variable to hold the time:
    }%

    do
     var TIME := 0 : integer in

      do
        every MS do
          TIME := TIME + 1
        end every
      upto STOP;

      emit DISPLAY(TIME);

      % Send the AVERAGE module the time via a integer signal:

      emit INCREMENT_AVERAGE(TIME)

    end var

    watching LIMIT_TIME MS timeout
     exit ERROR
    end timeout;

    emit GO_OFF;
    exit END_MEASURE

   ]
  end trap
end repeat;

% Final display:
```

```
    await PAUSE_LENGTH MS do

    %{
      Display the results of the average value that came from the AVERAGE
      module via its output signal:
    }%

    emit DISPLAY(?AVERAGE_VALUE)

   end await;
  exit END_GAME
  ]
 end signal

 handle ERROR do
    emit TILT_ON;
    emit GO_OFF
 end trap;

 emit GAME_OVER_ON

end every

end module
```

C Code

Reflex Game Esterel C Code.

reflex_game.c shows the resultant C output generated by the Esterel compiler.

main.c shows the C to Esterel interface.

```
/* occ : C CODE OF AUTOMATON REFLEX_GAME */

/* AUXILIARY DECLARATIONS */

#ifndef STRLEN
#define STRLEN 81
#endif
#define _COND(A,B,C) ((A)?(B):(C))
#ifdef TRACE_ACTION
#include <stdio.h>
#endif
#ifndef NULL
#define NULL ((char*)0)
#endif

#ifndef __EXEC_STATUS_H_LOADED
#define __EXEC_STATUS_H_LOADED

typedef struct {
unsigned int start:1;
unsigned int kill:1;
unsigned int active:1;
unsigned int suspended:1;
unsigned int prev_active:1;
```

```
unsigned int prev_suspended:1;
unsigned int exec_index;
unsigned int task_exec_index;
void (*pStart)();
void (*pRet)();
} __ExecStatus;

#endif
#define __ResetExecStatus(status) {\
   status.prev_active = status.active; \
   status.prev_suspended = status.suspended; \
   status.start = status.kill = status.active = status.suspended = 0; }
#define __DSZ(V) (--(V)<=0)
static int __REFLEX_GAME_engine();
#define __BASIC_ACT(i) (*__REFLEX_GAME_PActionArray[i])()
#ifdef TRACE_ACTION
#define __ACT(i) fprintf(stderr, "__REFLEX_GAME_A%d\n", i);__BASIC_ACT(i)
#else
#define __ACT(i) __BASIC_ACT(i)
#endif
#define BASIC_TYPES_DEFINED
typedef int boolean;
typedef int integer;
typedef char* string;
#define _true 1
#define _false 0
#define __REFLEX_GAME_GENERIC_TEST(TEST) {if (TEST) __REFLEX_GAME_cp++; else __REFLEX_GAME_cp +
typedef unsigned char  __REFLEX_GAME_indextype;
typedef void (*__REFLEX_GAME_APF)();
static __REFLEX_GAME_APF *__REFLEX_GAME_PActionArray;

#include "reflex_game.h"

/* EXTERN DECLARATIONS */

#ifndef _NO_EXTERN_DEFINITIONS
#ifndef _NO_CONSTANT_DEFINITIONS
#ifndef _LIMIT_TIME_DEFINED
#ifndef LIMIT_TIME
extern integer LIMIT_TIME;
#endif
#endif
#ifndef _MEASURE_NUMBER_DEFINED
#ifndef MEASURE_NUMBER
extern integer MEASURE_NUMBER;
#endif
#endif
#ifndef _PAUSE_LENGTH_DEFINED
#ifndef PAUSE_LENGTH
extern integer PAUSE_LENGTH;
#endif
#endif
#endif
#ifndef _NO_FUNCTION_DEFINITIONS
#ifndef _RANDOM_DEFINED
#ifndef RANDOM
extern integer RANDOM();
#endif
#endif
#endif
#endif
```

```
/* INITIALIZED CONSTANTS */

/* MEMORY ALLOCATION */

static boolean __REFLEX_GAME_V0;
static boolean __REFLEX_GAME_V1;
static boolean __REFLEX_GAME_V2;
static boolean __REFLEX_GAME_V3;
static integer __REFLEX_GAME_V4;
static integer __REFLEX_GAME_V5;
static integer __REFLEX_GAME_V6;
static integer __REFLEX_GAME_V7;
static integer __REFLEX_GAME_V8;
static integer __REFLEX_GAME_V9;
static integer __REFLEX_GAME_V10;
static integer __REFLEX_GAME_V11;
static integer __REFLEX_GAME_V12;
static integer __REFLEX_GAME_V13;
static integer __REFLEX_GAME_V14;


/* INPUT FUNCTIONS */

void REFLEX_GAME_I_MS () {
__REFLEX_GAME_V0 = _true;
}
void REFLEX_GAME_I_COIN () {
__REFLEX_GAME_V1 = _true;
}
void REFLEX_GAME_I_READY () {
__REFLEX_GAME_V2 = _true;
}
void REFLEX_GAME_I_STOP () {
__REFLEX_GAME_V3 = _true;
}

/* FUNCTIONS RETURNING NUMBER OF EXEC */

int REFLEX_GAME_number_of_execs () {
return (0);
}


/* AUTOMATON (STATE ACTION-TREES) */

static __REFLEX_GAME_indextype __REFLEX_GAME_sct0 [] = {
0,0
};
static __REFLEX_GAME_indextype __REFLEX_GAME_sct1 [] = {
20,10,11,14,8,0,2
};
static __REFLEX_GAME_indextype __REFLEX_GAME_sct2 [] = {
5,17,21,8,10,12,14,22,32,33,37,4,23,0,3,30,0,4,
0,2
};
static __REFLEX_GAME_indextype __REFLEX_GAME_sct3 [] = {
5,17,21,8,10,12,14,22,32,33,37,4,23,0,3,30,0,4,
4,13,39,6,10,13,11,0,2,7,2,15,0,3,7,4,15,0,3,
6,4,24,0,5,0,3
};
```

```
static __REFLEX_GAME_indextype __REFLEX_GAME_sct4 [] = {
5,17,21,8,10,12,14,22,32,33,37,4,23,0,3,30,0,4,
4,10,42,6,31,8,11,0,2,0,4,0,4
};
static __REFLEX_GAME_indextype __REFLEX_GAME_sct5 [] = {
5,17,21,8,10,12,14,22,32,33,37,4,23,0,3,30,0,4,
4,17,7,6,10,13,11,0,2,40,6,9,29,25,0,6,0,5,
7,6,10,13,11,0,2,6,2,15,0,5
};
static __REFLEX_GAME_indextype __REFLEX_GAME_sct6 [] = {
5,17,21,8,10,12,14,22,32,33,37,4,23,0,3,30,0,4,
4,28,41,6,10,13,11,0,2,7,16,27,8,28,34,35,36,10,
38,4,30,0,4,23,0,3,26,0,6,6,4,15,0,6,7,16,
27,8,28,34,35,36,10,38,4,30,0,4,23,0,3,0,6
};
static __REFLEX_GAME_indextype* __REFLEX_GAME_dct [] = {
(__REFLEX_GAME_indextype*)0 /* no-sub-dags */
};
static __REFLEX_GAME_indextype* __REFLEX_GAME_sct [] = {
__REFLEX_GAME_sct0,
 __REFLEX_GAME_sct1,
 __REFLEX_GAME_sct2,
 __REFLEX_GAME_sct3,
 __REFLEX_GAME_sct4,
 __REFLEX_GAME_sct5,
 __REFLEX_GAME_sct6
};

static __REFLEX_GAME_indextype* __REFLEX_GAME_cp = __REFLEX_GAME_sct1;

/* ACTIONS */

/* PREDEFINED ACTIONS */

static void __REFLEX_GAME_A1 () {
extern __REFLEX_GAME_indextype* __REFLEX_GAME_dct[];
__REFLEX_GAME_indextype* old_cp = __REFLEX_GAME_cp + 1;
__REFLEX_GAME_cp = __REFLEX_GAME_dct[*__REFLEX_GAME_cp];
__REFLEX_GAME_engine();
__REFLEX_GAME_cp = old_cp;
}
static void __REFLEX_GAME_A2 () {
extern __REFLEX_GAME_indextype* __REFLEX_GAME_dct[];
__REFLEX_GAME_cp = __REFLEX_GAME_dct[*__REFLEX_GAME_cp];
}
static void __REFLEX_GAME_A3 () {
__REFLEX_GAME_GENERIC_TEST(_false);
}

/* PRESENT SIGNAL TESTS */

static void __REFLEX_GAME_A4 () {
__REFLEX_GAME_GENERIC_TEST(__REFLEX_GAME_V0);
}
static void __REFLEX_GAME_A5 () {
__REFLEX_GAME_GENERIC_TEST(__REFLEX_GAME_V1);
}
static void __REFLEX_GAME_A6 () {
__REFLEX_GAME_GENERIC_TEST(__REFLEX_GAME_V2);
}
static void __REFLEX_GAME_A7 () {
```

```
__REFLEX_GAME_GENERIC_TEST(__REFLEX_GAME_V3);
}

/* OUTPUT ACTIONS */

static void __REFLEX_GAME_A8 () {
REFLEX_GAME_O_DISPLAY(__REFLEX_GAME_V4);
}
static void __REFLEX_GAME_A9 () {
REFLEX_GAME_O_GO_ON();
}
static void __REFLEX_GAME_A10 () {
REFLEX_GAME_O_GO_OFF();
}
static void __REFLEX_GAME_A11 () {
REFLEX_GAME_O_GAME_OVER_ON();
}
static void __REFLEX_GAME_A12 () {
REFLEX_GAME_O_GAME_OVER_OFF();
}
static void __REFLEX_GAME_A13 () {
REFLEX_GAME_O_TILT_ON();
}
static void __REFLEX_GAME_A14 () {
REFLEX_GAME_O_TILT_OFF();
}
static void __REFLEX_GAME_A15 () {
REFLEX_GAME_O_RING_BELL();
}

/* ASSIGNMENTS */

static void __REFLEX_GAME_A16 () {
__REFLEX_GAME_V0 = _false;
}
static void __REFLEX_GAME_A17 () {
__REFLEX_GAME_V1 = _false;
}
static void __REFLEX_GAME_A18 () {
__REFLEX_GAME_V2 = _false;
}
static void __REFLEX_GAME_A19 () {
__REFLEX_GAME_V3 = _false;
}
static void __REFLEX_GAME_A20 () {
__REFLEX_GAME_V4 = 0;
}
static void __REFLEX_GAME_A21 () {
__REFLEX_GAME_V4 = 0;
}
static void __REFLEX_GAME_A22 () {
__REFLEX_GAME_V7 = MEASURE_NUMBER;
}
static void __REFLEX_GAME_A23 () {
__REFLEX_GAME_V8 = LIMIT_TIME;
}
static void __REFLEX_GAME_A24 () {
__REFLEX_GAME_V9 = RANDOM();
}
static void __REFLEX_GAME_A25 () {
__REFLEX_GAME_V10 = 0;
```

```
}
static void __REFLEX_GAME_A26 () {
__REFLEX_GAME_V10 = __REFLEX_GAME_V10+1;
}
static void __REFLEX_GAME_A27 () {
__REFLEX_GAME_V4 = __REFLEX_GAME_V10;
}
static void __REFLEX_GAME_A28 () {
__REFLEX_GAME_V5 = __REFLEX_GAME_V10;
}
static void __REFLEX_GAME_A29 () {
__REFLEX_GAME_V11 = LIMIT_TIME;
}
static void __REFLEX_GAME_A30 () {
__REFLEX_GAME_V12 = PAUSE_LENGTH;
}
static void __REFLEX_GAME_A31 () {
__REFLEX_GAME_V4 = __REFLEX_GAME_V6;
}
static void __REFLEX_GAME_A32 () {
__REFLEX_GAME_V13 = 0;
}
static void __REFLEX_GAME_A33 () {
__REFLEX_GAME_V14 = 0;
}
static void __REFLEX_GAME_A34 () {
__REFLEX_GAME_V13 = __REFLEX_GAME_V13+__REFLEX_GAME_V5;
}
static void __REFLEX_GAME_A35 () {
__REFLEX_GAME_V14 = __REFLEX_GAME_V14+1;
}
static void __REFLEX_GAME_A36 () {
__REFLEX_GAME_V6 = __REFLEX_GAME_V13/__REFLEX_GAME_V14;
}

/* PROCEDURE CALLS */

/* CONDITIONS */

static void __REFLEX_GAME_A37 () {
__REFLEX_GAME_GENERIC_TEST(__REFLEX_GAME_V7>0);
}

/* DECREMENTS */

static void __REFLEX_GAME_A38 () {
__REFLEX_GAME_GENERIC_TEST(__DSZ(__REFLEX_GAME_V7));
}
static void __REFLEX_GAME_A39 () {
__REFLEX_GAME_GENERIC_TEST(__DSZ(__REFLEX_GAME_V8));
}
static void __REFLEX_GAME_A40 () {
__REFLEX_GAME_GENERIC_TEST(__DSZ(__REFLEX_GAME_V9));
}
static void __REFLEX_GAME_A41 () {
__REFLEX_GAME_GENERIC_TEST(__DSZ(__REFLEX_GAME_V11));
}
static void __REFLEX_GAME_A42 () {
__REFLEX_GAME_GENERIC_TEST(__DSZ(__REFLEX_GAME_V12));
}
```

```
/* START ACTIONS */

/* KILL ACTIONS */

/* SUSPEND ACTIONS */

/* ACTIVATE ACTIONS */

/* WRITE ARGS ACTIONS */

/* RESET ACTIONS */

/* ACTION SEQUENCES */

/* THE ACTION ARRAY */

static __REFLEX_GAME_APF __REFLEX_GAME_ActionArray[] = {
0,
(__REFLEX_GAME_APF)__REFLEX_GAME_A1,
(__REFLEX_GAME_APF)__REFLEX_GAME_A2,
(__REFLEX_GAME_APF)__REFLEX_GAME_A3,
(__REFLEX_GAME_APF)__REFLEX_GAME_A4,
(__REFLEX_GAME_APF)__REFLEX_GAME_A5,
(__REFLEX_GAME_APF)__REFLEX_GAME_A6,
(__REFLEX_GAME_APF)__REFLEX_GAME_A7,
(__REFLEX_GAME_APF)__REFLEX_GAME_A8,
(__REFLEX_GAME_APF)__REFLEX_GAME_A9,
(__REFLEX_GAME_APF)__REFLEX_GAME_A10,
(__REFLEX_GAME_APF)__REFLEX_GAME_A11,
(__REFLEX_GAME_APF)__REFLEX_GAME_A12,
(__REFLEX_GAME_APF)__REFLEX_GAME_A13,
(__REFLEX_GAME_APF)__REFLEX_GAME_A14,
(__REFLEX_GAME_APF)__REFLEX_GAME_A15,
(__REFLEX_GAME_APF)__REFLEX_GAME_A16,
(__REFLEX_GAME_APF)__REFLEX_GAME_A17,
(__REFLEX_GAME_APF)__REFLEX_GAME_A18,
(__REFLEX_GAME_APF)__REFLEX_GAME_A19,
(__REFLEX_GAME_APF)__REFLEX_GAME_A20,
(__REFLEX_GAME_APF)__REFLEX_GAME_A21,
(__REFLEX_GAME_APF)__REFLEX_GAME_A22,
(__REFLEX_GAME_APF)__REFLEX_GAME_A23,
(__REFLEX_GAME_APF)__REFLEX_GAME_A24,
(__REFLEX_GAME_APF)__REFLEX_GAME_A25,
(__REFLEX_GAME_APF)__REFLEX_GAME_A26,
(__REFLEX_GAME_APF)__REFLEX_GAME_A27,
(__REFLEX_GAME_APF)__REFLEX_GAME_A28,
(__REFLEX_GAME_APF)__REFLEX_GAME_A29,
(__REFLEX_GAME_APF)__REFLEX_GAME_A30,
(__REFLEX_GAME_APF)__REFLEX_GAME_A31,
(__REFLEX_GAME_APF)__REFLEX_GAME_A32,
(__REFLEX_GAME_APF)__REFLEX_GAME_A33,
(__REFLEX_GAME_APF)__REFLEX_GAME_A34,
(__REFLEX_GAME_APF)__REFLEX_GAME_A35,
(__REFLEX_GAME_APF)__REFLEX_GAME_A36,
(__REFLEX_GAME_APF)__REFLEX_GAME_A37,
(__REFLEX_GAME_APF)__REFLEX_GAME_A38,
(__REFLEX_GAME_APF)__REFLEX_GAME_A39,
(__REFLEX_GAME_APF)__REFLEX_GAME_A40,
(__REFLEX_GAME_APF)__REFLEX_GAME_A41,
(__REFLEX_GAME_APF)__REFLEX_GAME_A42
};
```

```
static __REFLEX_GAME_APF *__REFLEX_GAME_PActionArray  = __REFLEX_GAME_ActionArray;


static void __REFLEX_GAME__reset_input () {
__REFLEX_GAME_V0 = _false;
__REFLEX_GAME_V1 = _false;
__REFLEX_GAME_V2 = _false;
__REFLEX_GAME_V3 = _false;
}

/* AUTOMATON ENGINE */

static int __REFLEX_GAME_engine () {
register __REFLEX_GAME_indextype x;
while (x = *(__REFLEX_GAME_cp++)) {
__ACT(x);
}
return *__REFLEX_GAME_cp;
}

int REFLEX_GAME () {
int x;
x = __REFLEX_GAME_engine();
__REFLEX_GAME_cp = __REFLEX_GAME_sct[x];
__REFLEX_GAME__reset_input();
return x!=0;
}

/* AUTOMATON RESET */

int REFLEX_GAME_reset () {
__REFLEX_GAME_cp = __REFLEX_GAME_sct1;
__REFLEX_GAME__reset_input();
return 0;
}
```