DESIGN ABSTRACTIONS FOR PROTOCOL SOFTWARE

Paul Taylor, VK3BLY
P.O. Box 118
Eltham, 3095
Victoria
Australia

Abstract

As amateur packet radio software becomes more complicated and software develoment environments improve, the use of high level languages will become more favourable. A design approach for protocol software based on modules and Finite State Machines is described, which formalises the interface to IO devices, and extends the use of the protocol's State Machine model into the implementation stage. Its adoption should make implementations of Level 2 and 3 protocols quicker, easier and more understandable.

Introduction

Traditionally, communication software for microprocessors was written in the native assembly language, for two reasons; the simple microprocessor architectures limited the code complexity, size and speed, and the environment needed for software development at a higher level was not available. The early Terminal Node Controllers were implemented under these restrictions.

The first objection is no longer valid. The explosion of the personal computer has made significant processing power cheap and readily available, and many of these machines can be used for both software development and as a dedicated or shared host.

Secondly, software tools and environments have grown into a rich and Plentiful arena to develop software in. Compilers, interpreters, simulations, graphics, databases and communications software of surprising complexity and usefulness are now available for even the simplest of microprocessors.

Protocol software developers should consider adopting high level languages (such as Pascal and C) for their communications systems. The advantages of expressing software in a structured, high level language are well known (see any Software Engineering text).

Once a high level language has been chosen, some important design considerations must be faced. The features of the chosen language must be exploited to improve the software design, implementation and maintenance. Put more generally, the problem may be worded as a question; how should a communications program be designed? A design approach which answers this question will be discussed.

Data Abstraction

All software for driving devices (ie. ports) should be modularised. High level languages offer mechanisms for successive abstraction or modularisation, usually in the form of procedures/functions or subroutines. A design approach enforces the decomposition of software into modules in a disciplined way.

Modularisation allows for the implementation of well defined interfaces to entire functionally-independent components of a software system. A module (within a program) is one or more sub-programs which perform a specified task on some data. The module encapsulates both the data object, and the sub-programs which manipulate it. All aspects of the module are completely defined by the programmer, in an appropriate notation.

Definition of a Module

An example drawn from packet radio software is shown below. Modules are defined for a serial port (ie. a UART), and a "packet_port" (ie. the software interface to a protocol controller chip),

These definitions are presented in a Pascal-like pseudocode, a form which has been successfully used by the author. The purpose of the notation is to convey semantic specification, rather than syntactic detail.

```
serial_port:
 {global data}
 registers:
   begin
     TXbuff, RXbuff, DivisorLSB,
     DivisorMSB, Intrpt, LineControl,
     LineStat, ModemControl,
     ModemStat:UART_Register;
   end
 {operators }
   procedure initialise;
   procedure send_char (ch:char);
   function char_received:boolean;
   function get_received_char:char;
end {serial_port};

packet_port:
   {global data}
   buffers:array[1..n] of buffer_type;
   registers:
    begin
      TXbuff, RXbuff, Intrpt, LineControl,
      LineStat, ModemControl, ModemStat
      :Protocol_Controller_Register;
    end
   {operators }
   procedure initialise;
   procedure send_frame(b:buffer);
   procedure listen(b:buffer);
   function frame_received:boolean;
   procedure get_received_frame(
                        var b:buffer);
end {packet_port};
```

Once the module (port interface) has been specified as the examples show, implementation of the module can proceed by further refining the data objects, and by writing the interface procedures or function code. If necessary, the module specification can be modified slightly in the light of implementation considerations, but this should be avoided since it implies weakness in the original module specification.

Once implemented, using the module should be a simple case of making calls to the interface procedures or functions as required. Since languages such as Pascal and C do not support modules explicitly, it is the programmer's responsibility to enforce correct use of the module in his or her code, by using the defined interfaces only. For instance, when using a Packet-port module, there must be no reading or writing of any of the data variables, or the port device itself.

The advantages and disadvantages of using this organisation of device interface software will be discussed later.

Control Abstraction

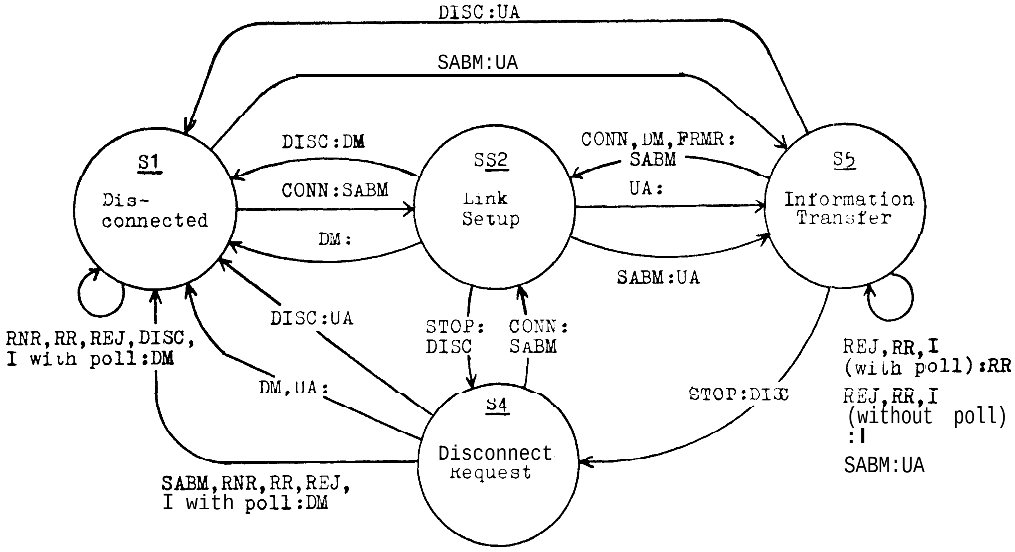
Communication systems are usually "event driven"; the software must respond to asynchronous events, such as an operator keyed command, or a received packet. Such software must decide on appropriate processing of events on the basis of the history, or state of the system. A formal model which is applicable in such systems is the Finite State Machine.

A Finite State Machine (FSM) consists of a finite set of states, each of which knows of a set of events, and a transition function which maps (state, event) into (new-state, action). The action is a member of a set of actions, which express interaction or modification of the FSM's environment.

FSMs are easily depicted in diagrammatic form. The states are represented by circles, transitions between states by directed arcs, and events and actions by "event:action" labels on the transitions. Part of the FSM model of AX25 [Fox 84] is presented in Figure 1. It shows the states which are passed through when the local station connects to a remote station.

The following steps take place when a connection is established. Initially, our station is in the "Disconnected" state. Our attempt to connect to another station (by typing "CONN <callsign>") is interpreted by the FSM as an event, and classified (e19) The FSM state transition function is invoked, with parameters current-state (which is "Disconnected"), and event (which is e19). The function yields a new state ("Link Establishment"), and an action (send a SABM packet to the addressed station). The action is performed and the system waits, ready to respond to any of the events which may occur in the "Link Establishment" state.

From this simplified example, it should be clear that an FSM model is ideal for specification, design and implementation of an event driven system, such as a protocol. The advantages of adoption of the FSM model are discussed later. The rest of this section describes some ideas on how FSMs can be implemented in Pascal, a representative high level language.



Note: CONN and STOP are operator commands for connecting and disconnecting to the remote station respectively.
Al.1 other names are AX.25 frame types.

AX.25 FSM EVENT / STATE	e0 I with Poll	e8 SABM either	e9 DISC either	e16 UA either	e17 DM either	e19 CONN cmd
1 Disconnected	DM	UA,S5	DM	DM	DM	SABM,S2
2 Link. Setup	-	UA,S5	DM,S1	-	-	
3 Frame Reject	FRMR	UA,S5	UA,S1	-		SABM,S2
4 Disconnect Rqst	DM,S1	DM,S1	UA,S1	S1	S1	SABM,S2
5 Information Xfer	R R	UA	UA,S1	-	-	SABM,S2
. .						
.	.	.	.	.	.	.

Figure 1. Extract of the AX.25 FSM.

FSM based Implementation of AX. 25

Once the FSM model of the protocol is completely defined, implementation becomes straightforward and somewhat, mechanical. The components to be defined are:

i. the state table,

ii. the state variable,

iii. the action procedures,

iv. the FSM procedure,

v. the mainloop.

Some examples of these follow. The Finite State Table (FST) can easily be defined:

```
const
   S1_Disconnected=1;
   S2_LinkEstablishment-2;
       .
   A0_NoAction=0;
   A1_SendSABM=1;
       .
   E1_CONNcmd=1;
   E2_DMreceived=2;
       .
   nbr_states =n;
   nbr_events =n ;
   nbr_actions=n;

type
   state_type  =
          S1_Disconnected..Sn_StateN;
   event-type  =
          E1_CONNcmd..En_EventN;
   action-type  =
          A0_NoAction..An_ActionN;
```

```
var
   FST:array[1..nbr_states,
             1..nbr_events] of
             record
                newstate:state_type;
                action  :action-type
             end;
```

The state variable is simply defined:

```
var  current_state:state_type;
```

Each (newstate, event) pair in the FST must be initialised. In Pascal, this becomes rather tedious:

```
FST[S1_Disconnected,
    E1_CONNcmd].newstate:=
           S2_LinkEstablishment;
FST[S1_Disconnected,
    E1_CONNcmd].action  :=
           A1_sendSABM;

FST[S2_LinkEstablishment,
    E2_DMreceived].newstate:=
           S1_Disconnected;
FST[S2_LinkEstablishment,
    E1_DMreceived].action  :=
           A0_NoAction;

      etc
       .
```

The state transition function can be implemented as a simple Pascal function:

```
function FSM(event:event_type):state_type;
{ globals: FST, current_state }
begin
  case FST[current_state, event], action of
  A0_NoAction: {do nothing}
  A1_sendSABM : begin sendSABM (,,..,,) end;
  A2_Action2 : begin { do Action 2 } end;
      .
  An_ActionN : begin { do Action n } end;
  end {case};
  FSM :=
    FST[current_state, event].newstate;
end {FSM};
```

The action procedures A0_NoAction ... An_ActionN take care of all Processing needed to handle the event. These Procedures may call other 'utility' procedures to do lower level processing.

The main loop of the Process takes the following form:

```
current-state := initial-state;
repeat
   get-event (event);
   current_state := FSM(event);
until current-state = terminal_state;
```

The procedure "get_event (event)" monitors both the packet_port, and the keyboard, for any input which can be classified as an event:

```
procedure get_event (var event:event_type);
var
   kbd_buffer:string[1..longest_cmd];
   frame_buffer:frame;

event := no-event ;
repeat
  { wait for received frame or typed key }
   if Key Pressed then
      get char
      if char is a CR then
         analyse_cmd(kbd_buffer,event)
      else
         append char to kbd_buffer;
  else
      if frame_received then
         analyse_frame(frame_buffer,event)
until event <> no_event;
```

The comment "wait on a received frame or typed key" refers to an implementation specific issue. If both frame reception and keyboard monitoring is done by interrupt handlers, Procedure get-event should unschedule itself at this Point. In an implementation where interrupts are not available, the procedure can simply loop until a received frame is detected or a key is pressed.

Experience with Design Abstractions

Experience has shown that adoption of these design abstractions has many more advantages than disadvantages.

Object-based implementations of device interface software (such as the serial_port or packet_port) offer all the advantages of modularity:

i. Abstraction. Specific details of the physical port device are hidden.

ii. A well defined interface. All references to the device go through a common interface.

iii. Interface definitions can be application specific. The definition of the interface to the Port (ie. the access Procedures) is entirely up to the programmer.

iv, Isolation. Replacement or upgrading of the physical device can be done transparently to the rest of the software.

A possible disadvantage is the overhead of structure. Using structure imposes the need for more object code and data, which may in some cases result in slower execution speed. This has not been a problem in both RTTY and AX.25 software, but may cause problems at higher baudrates.

The advantages of adopting the FSM model in implementation were found to be:

i. Centralisation of Control. The entire 'control policy' of the protocol is described neatly and simply in a single table.

ii. Simplicity. A complicated protocol can become readable and understandable.

iii. Enforcement of Structure. Use of an FSM model enforces a uniform approach to the organisation of the entire program.

iv. Modifiability. Maintenance of a piece of software is made easier by the strict enforcement of structure. Once the Finite State Table has been set up, and the set of action procedures written as drivers for lower level "utility procedures", new actions, transitions and states can usually be added quickly and easily.

The only disadvantage encountered was the potential for the FSM model to become complex. When protocols become complicated, their FSM model can become very large and cumbersome (this is called "state explosion"). Tabular representations of large FSMs can be managed, but graphical representations may be unwieldly.

Conclusion

The average personal computer can host a perfectly satisfactory environment for communication software development in a high level language.

The use of a high level language requires a more formal approach to software design. Software for driving the IO devices can conveniently be organised as a module. In an event driven system, the entire system's control flow can be encapsulated in a Finite State Machine model,

These design abstractions have been applied in the development of software for RTTY and AX.25 by the author. They were found to increase software reliability, and make the source code more understandable and modifiable.

References

[Fox 84] Terry L. Fox (WB4JFI), "AX.25 Amateur Packet-Radio Link-Layer Protocol", October 1984.