IAR Application Note #49597

Simple I/0 functions and visualSTATE

The information in this document is based on versions 4.2 and 4.3 of the IAR visualSTATE software. It may also apply to other versions of IAR visualSTATE.

SUMMARY

This application note describes the creation of a simple visualSTATE application that connects buttons to LEDs on an Atmel STK300 development board. Several important constructs are used such as *guard expressions*, *hierarchy*, and *timer action functions*. This application note illustrates how to create and use inputs-to-event functions, FIFO queues, and actions-to-output functions for communicating real-world events to/from the visualSTATE engine.

KEYWORDS

visualSTATE, periodic events, internal reactions, guard expressions, hierarchy, timer action functions, FIFO queues, visualSTATE engine.

The problem to be solved

The buttons and LEDs of an Atmel STK300 evaluation board must be made to work as a simple light switch, a flashing light blinking at approximately 2 Hz, a “3 times” light switch, and a “flashing bank” of LEDs.

The exact specifications of the application are:

1 Pressing Button1 on the Atmel STK300 should toggle LED1. If LED1 is off when Button1 is pressed, it should turn on.

2 Pressing Button2 should illuminate LED0 continually and should cause LED2 to flash at approximately 2 Hz. Pressing Button2 again should extinguish both LEDs.

3 Pressing Button3 should toggle LED3 in the same fashion as the Button1/LED1 combination. However, after the third complete on/off cycle of LED3, Button3 shall have no effect.

4 Pressing Button4 shall cause LEDs 4-7 to cycle in the 4->7 direction three times, and then in the 7->4 direction 3 times.

The difficulties involved

Functions must be created to capture real-world inputs such as button presses and place the corresponding events into a FIFO queue. Functions to make real-world outputs such as LED illumination must be created. The FIFO queue must be managed. Timers must be started to run periodic events. The application developer must create code to call the appropriate visualSTATE API functions to process events and execute actions.

Solution / Getting started

The first step is to create a visualSTATE model of the application. See *IAR visualSTATE Quick Start Tutorial* for a description of how to set up a visualSTATE Project.

Files used for the application

The files listed below are required for the application which will run on an Atmel STK300 development board (the application can easily be ported to a less resource-rich Atmel STK200 board as well).

The file containing the `main()` and functions that support I/O, timers, initialization	`port2STK300.c`
The project file for the Atmel STK300 development board	`portSTK300.prj`

visualSTATE design files	`STK300.nwp`
	`STK300.vsp`
	`Topstate.vsr`

visualSTATE Coder-generated files	`STK300.c`
	`STK300Data.c`
	`SEMBDef.h`
	`SEMTypes.h`
	`STK300.h`
	`STK300Data.h`
	`STK300Action.h`

visualSTATE Basic API files	`SEMLibB.c`
	`SEMLibB.h`
	`VSTypes.h`

Consult *IAR visualSTATE Quick Start Tutorial* for header and c files that must be in the project directory.

The statechart diagrams

Your statechart diagram is best divided into concurrent regions that handle the respective portions of the application. In this example the regions are `Reset`, `Left`, `Right`, `Test` and `motor`.

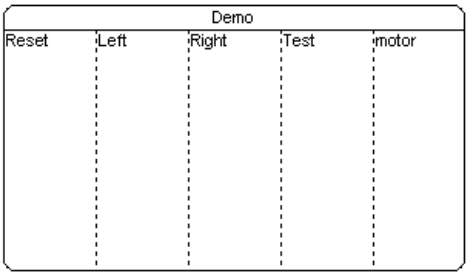

Figure 1: Statechart diagram with concurrent regions

There should be an initial state that the system enters upon the event SE_RESET. See *Figure 2*.

Figure 2: Initial state (contained in the region `Reset`*)*

As we will see later, this will be the first event put into the queue. Assigning the action function `LEDALL_OFF()` ensures that the hardware will be in a known, standard state at the beginning.

Then we will tackle the simplest specification first, namely the light switch where Button1 will toggle LED 1. This is shown in *Figure 3*.

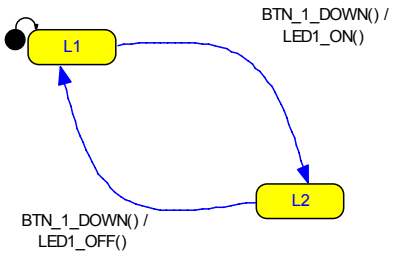

Figure 3: A simple light switch application (contained in the region `Left`*)*

The design consists of two states, `L1` and `L2` that are connected by transitions with events that correspond to pressing Button1. In this design, the LED toggle is implemented explicitly by the driver functions `LED1_ON()` and `LED1_OFF()`.

At this point it is worthwhile considering how a physical keypress is communicated to the visualSTATE engine. *Listing 1* shows the code that actually detects whether an up/down or a down/up transition of a button has occurred. In this example, code supporting both transitions is included although only up/down transitions result in occurrence of events that cause state changes. Although down/up transition events are placed in the queue, they are discarded by visualSTATE because they do not exist in the statechart model. This is an example of using general purpose driver functions that have more capability than the application actually requires.

```
Listing 1. Button Scanning Input to Event Function
void scan_buttons(void)
/*************************************************************************

 * This function scans the keypad for up -> down and *
 * down -> up transitions *
 * It can be used to detect either of *
 * those, or to do rudimentary debouncing. *
 * On the Atmel STK200 and STK300, 8 buttons are *
 * connected to PORTD Pins on PORTD are driven
 * LOW as long as the button is held down. *
  *************************************************************************/
{
  static UCHAR lastscan;
  volatile UCHAR thisscan;                /* don't let compiler optimize away */
                                          /* this statement */
  thisscan = ~PIND;

  if (thisscan != lastscan)               /* skip if button in same */
                                          /* position as last scan */
    if ((thisscan ^ lastscan) & 0x01)     /* either down->up or */
                                          /* up-> down happened */
#ifdef BTN_0_DOWN
      if (thisscan & 0x01)
        put_event(BTN_0_DOWN);            /* put button down event */
#endif                                    /* into queue */
#ifdef BTN_0_UP
      if (~thisscan & 0x01)
        put_event(BTN_0_UP);              /* put button up event*/
#endif                                    /* into queue */
    }

    if ((thisscan ^ lastscan) & 0x02)
    {
#ifdef BTN_1_DOWN
      if (thisscan & 0x02)
        put_event(BTN_1_DOWN);
#endif
#ifdef BTN_1_UP
      if (~thisscan & 0x02)
        put_event(BTN_1_UP);
#endif
    }

...similar logic for  buttons 2-7...

   lastscan = thisscan;                   /* remember if up/down or */
  }                                       /*down/up transition */
}
```

Listing 1

The code shown in *Listing 1* puts a button down or a button up event into the queue, depending on which input it detects. It will only put one event into the queue for each transition because of the test `thisscan != lastscan`. In the particular architecture of the Atmel STK300, a pin on PORTD is driven low as long as the corresponding button is held down. Although there is no explicit debouncing built into the keypad service routine, it could easily be added.

The action functions `LED1_ON()` and `LED1_OFF()` are called during the transitions from states L1 to L2. Code that supports this on the Atmel STK300 is shown in *Listing 2*.

```
Listing 2 - Drivers for on board LEDs.
/*********************************************************************
 In this section use is made of the fact that driving a pin LOW on PORTB of
STK300 will turn on the LED.
*********************************************************************/
void LED0_ON(void)
{
  PORTB &= ~0x01;
}

void LED0_OFF(void)
{
  PORTB |= 0x01;
}
...etc. for other LEDs on the board
```

Listing 2

Periodic events, timers and interrupt service routines

The heart and soul of embedded applications is making things happen at a periodic interval. The specifications above call for LED2 to be a “heartbeat” indicator that flashes at a 2 KHz rate. In this section, we will see how to the visualSTATE-generated code for the statechart model should be interfaced with the user-written code needed for service timer interrupts.

Internal reactions

In *Figure 4* we see that the state `R1` is exited and state `R2` is entered when the event corresponding to pressing Button2 is processed.

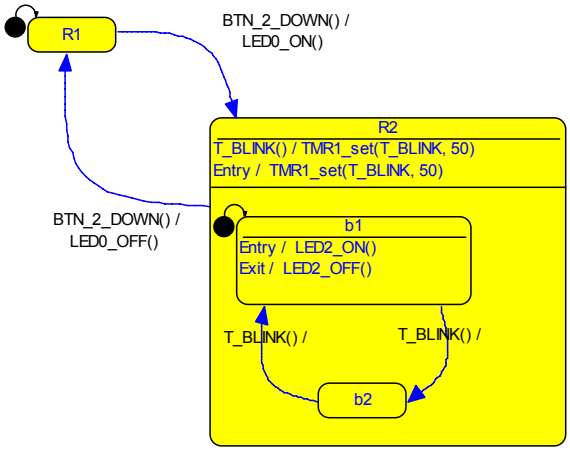

Figure 4: Example of internal reaction (state machine contained in the region `Right` *)*

Upon entry into state R2, the entry reaction containing the timer action function TMR1_SET is executed. The purpose of this function, TMR1_set(T_BLINK,50) is to create a timer object that will, in this case, tick down for 50 system ticks and then launch the T_BLINK() event by inserting it into the FIFO queue. To keep this process running, we will use an internal reaction with the T_BLINK event as a trigger.

An internal reaction is a transition that fires inside the state, in this case R2. An internal reaction is used because we do not want to exit R2 and return to it. Rather, we want transitions to fire while we are in the state. Upon entering R2, notice that state b1, which is a substate of R2, is the initial state.

Consequently, upon entry into state R2, the system will also be in state b1. Upon entry into this state, the action function LED2_ON() will be executed. The system will "park" in b1 until the T_BLINK() event takes it to state b2. However, T_BLINK() does not occur until Timer 1 expires. Timer 1 was installed by calling the action function TMR1_set(T_BLINK,50). In this case, 50 is the number of system ticks that the timer will count down.

The application developer must make the connection between the number of system ticks, and the actual number of milliseconds or seconds that the timer needs to tick down before it expires. This could be implemented by using

- A data structure that describes the individual timers,
- An interrupt service routine that knows how to manage the timer objects,
- Some code in main() that knows how and when to scan the timer objects and put events into the queue upon timeout.

Listing 3 shows the data structure for the timer objects. The structure holds two arrays:

- One array holds the number of timer ticks that the individual timer is to decrement during countdown.
- The other array holds the event that will be put into the queue when the timer expires.

```
Listing 3 Timer object structure
struct timers_tag
{
  VS_UINT timer[NOF_TIMERS]; /* A global timer variable*/
  VS_UINT   timer_event[NOF_TIMERS];
}Timer;
```

Listing 3

Listing 4 shows how the timer is installed. The function TMR1_set() takes a parameter for the event to be launched at timeout, and the number of ticks to be counted down. In the main() function there is a for() loop that steps through the timer objects. If they have timed out, the statement put_event(Timer.timer_event[i]); inserts the event into the queue. On each traversal through the timer interrupt service routine, the timer object structure element that holds the number of system ticks is decremented. In this example, Timer0 of the Atmel Mega103 chip is used as the system timer. This is an 8-bit timer. The external oscillator runs at 4 MHz. If a prescalar value of 256 is chosen, then the interrupt happens every 8.2 ms. In this case, 50 timer ticks represents 410 ms, making LED2 blink at 2.4 Hz. To make it blink at exactly 2 Hz, use 61 timer ticks in TMR1_set().

```
Listing 4. Timer installation function
void TMR1_set(VS_UINT event,VS_UINT ticks)
{
  Timer.timer[0] = ticks;
  Timer.timer_event[0] = event;
}
```

Listing 4

Listing 5 and *Listing 6* are the timer scan loop in `main()` and the timer interrupt service routine respectively.

```
Listing 5 Timer scan loop in main() that signals timeout event to
visualSTATE engine
for (i = 0; i < NOF_TIMERS; i++)  /* Check timers for timeout   */
{
    if (Timer.timer[i] == 1)
    {
        Timer.timer[i] = 0;
        put_event(Timer.timer_event[i]);
        Timer.timer_event[i] = EVENT_UNDEFINED;
    }
}
```

Listing 5

```
Listing 6 Timer Interrupt Service Routine
interrupt [TIMER0_OVF_vect] void TIMER0_interrupt(void)
/*****************************************************
 *TIMER0 OVERFLOW INTERRUPT HANDLER       *
 * caveat: This application will work for 1.1 years at 1 tick/8.2*
 * msec before timer events firing because of rollover problems*
 *****************************************************/
{
  ul_tick++;
  if (Timer.timer[0] > 1)
    Timer.timer[0]--;
  if (Timer.timer[1] > 1)
    Timer.timer[1]--;
}
```

Listing 6

Guard expressions

As a tutorial example, we design a light switch that breaks after three tries. This kind of paradigm is typical in security systems that may, for example, limit the number of attempted card reads in an access control system.

In our example, the application looks very much like the simple light switch, except that the guard expression `a<3` is specified on the left side (condition side) of the transition, and the assignment `a=a+1` is specified on the right side (action side) of the transition. This means that event `BTN_3_DOWN()` will cause a transition to fire if and only if the guard expression is satisfied, that is `a<3`. When the transition fires, the value of `a` is incremented according to the expression `a=a+1`.

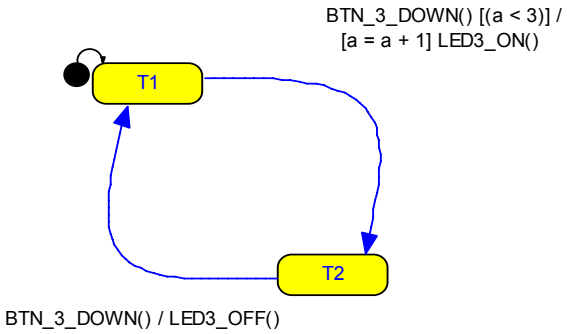

Figure 5: Model contained in the region `Test`

Although further `BTN_3_DOWN()` events go into the queue if Button3 is pressed, they are subsequently discarded by visualSTATE during the deduction process. This effectively "ignores" Button3 presses forever afterwards.

Multiple timers and psuedoconcurrent behavior

The model in the `motor` region of the system makes use of some of the constructs that we have discussed above.

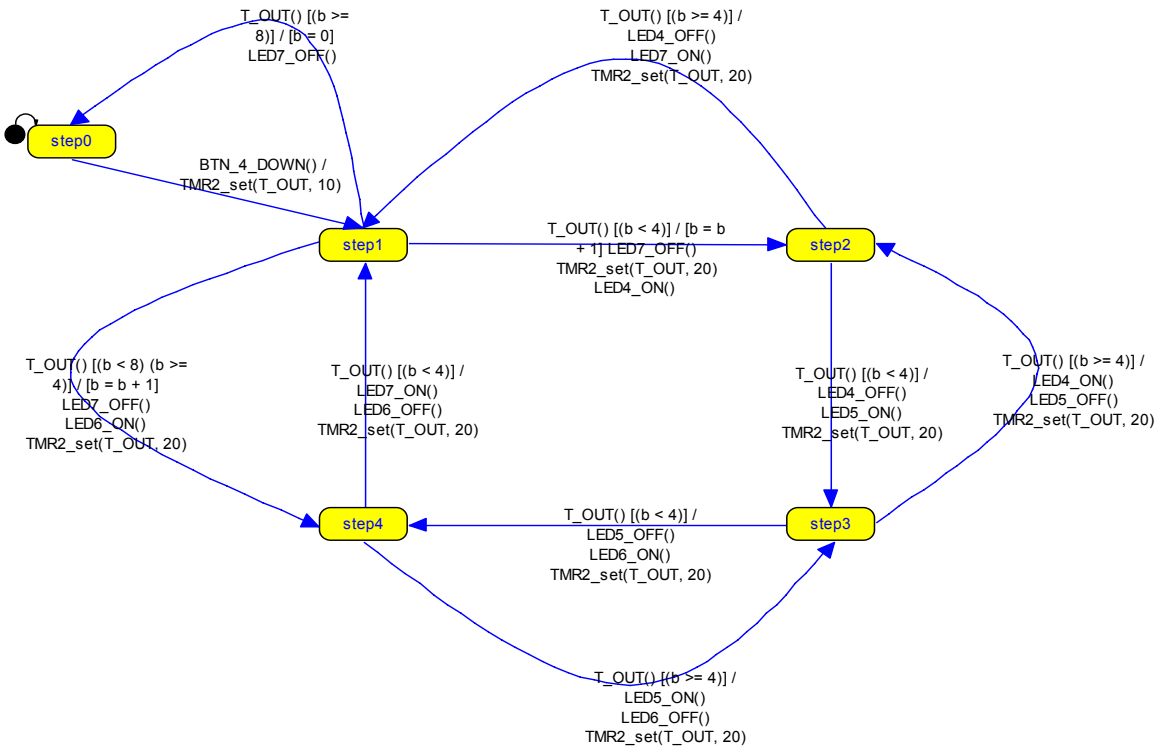

Figure 6: Model contained in the region `motor`

The model in the `motor` region installs another timer, `TMR2` to handle timing of events independent of `TMR1`. However, the mechanism by which the timer is set up, counts down, and launches events is exactly the same as `TMR1`. Operating the state machines in the two regions `Right` and `motor` by pressing Button2 and then Button4 respectively shows behavior where the state machines appear to be operating concurrently.

The main() function and visualSTATE API

The application developer must supply a `main()` function which accomplishes the following tasks:

1 Actualizes the control logic by processing events in the queue.

2 Scans all input devices (in this case the buttons on the Atmel STK300).

3 Scans the timer objects for timed out situations.

Listing 7 shows the `main()`. The first task is to create the scratch variable `output` that is needed by visualSTATE. Second, the peripheral devices on the board are initialized. A call to the global initialization function initializes all global variables. In this case it includes the queue and the timer objects. After this, calls are issued to the appropriate visualSTATE API functions to initialize the state machine and variables. The SE_RESET and SE_INIT events are manually inserted in the queue by calls to the queue functions. The `starttimer()` function handles low-level details of starting up Timer0 on the Mega103 chip and enabling timer and global interrupts.

Listing 7. Code that implements visualSTATE engine

```
void main(void)
{
    SEM_ACTION_EXPRESSION_TYPE output;

    init_STK300_registers();                  /* setup I/O ports */
    init_global_variables();                  /* Initialize all global variables */

    SEM_Init();                               /*  INITIALIZE SEM DATA  */
    SEM_InitInternalVariables();

    put_event(SE_RESET);
    put_event(SE_INIT);

    starttimer(TIMER0, PRESCALAR_128);        /*start timer tick */
                                              /*every 8.2 msec */
    while(1)                                  /*  MAIN LOOP OF DEMO PROGRAM  */
    {
         if (any_event())
         {
              SEM_Deduct(get_event());
              while(SEM_GetOutput(&output)==SES_FOUND)
                   SEM_Action(output);
              SEM_NextState();
         }
         for (i = 0; i < NOF_TIMERS; i++)
         {
              if (Timer.timer[i] == 1)
              {
                   Timer.timer[i] = 0;
                   put_event(Timer.timer_event[i]);
                   Timer.timer_event[i] = EVENT_UNDEFINED;
              }
         }
```

```
        if ((ul_tick - ul_lasttick) >= TICKS_PER_50MSEC)
        {
             scan_buttons();
             ul_lasttick = ul_tick;
        }

    }                                          /* close while(1) loop */
}                                              /*  end main  */
```

Listing 7

Event processing and FIFO queues

Event processing begins with the `if(any_event())`. If there are any events in the queue, the following visualSTATE Basic API functions for event processing are called:

- The `SEM_Deduct()` function prepares the visualSTATE System to execute the appropriate actions based on the given event, the internal state vector and the rules in the current visualSTATE System.
- `SEM_GetOutput()` finds the next action expression that is to be executed, based on the current event, the state of the visualSTATE System and the current rules.
- `SEM_Action()` is a macro that will call the current action function by using an internal visualSTATE function pointer table.
- `SEM_NextState()` updates the internal current state vector if any states can be found, based on the event given to `SEM_Deduct()` and the current state of the visualSTATE System.

This process occurs as long as there are events in the queue to be processed, that is, when `any_event() == TRUE`.

FIFO queues play an important role in the successful implementation of a visualSTATE application. *Figure 7* depicts the interaction of the visualSTATE engine with the event queue.

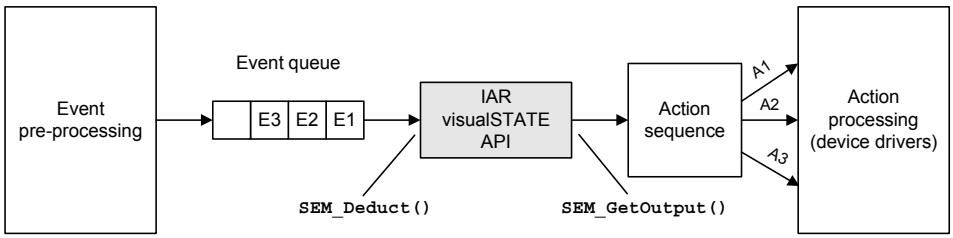

Figure 7: Conceptual view of visualSTATE engine

Queues are the common insertion point of events regardless of source. In this application note, we have seen events inserted into the buffer by I/O drivers using statements such as `put_event(BTN_0_DOWN);` or by expired timers such as `put_event(Timer.timer_event[i]);`.

At minimum, we need three queue functions:

- One for putting events into the queue
- One for fetching events from the queue
- One for checking if there is at least one event in the queue.

Further error handling (such as queue-full situations) is the responsibility of the application developer.

Using FIFO queues to buffer inputs is a common technique in embedded systems programming. Inserting events into queues is a general method of deterministic scheduling. If events of extremely high priority must be handled, it is possible to insert them directly into the head of the queue.

Code space

The application in this example is not highly optimized for code space. For example, we write explicit drivers for each LED on the board. This is useful for documentation purposes—looking at the statechart tells you exactly what is going on, for example in region `Left`, `LED1` is turned on by pressing Button1.

Code space usage for LED drivers is reduced by parameterizing the LED drivers, but using these kinds of functions would be less self-documenting than explicit driver functions.

The `motor` region of the statechart could have a much more economical design. We have included this design for tutorial purposes, but it is an interesting exercise to express the same functionality in a more concise way.

The keypad scanning routine stops short of full debouncing. Rudimentary debouncing exists because the keypad is only scanned at 50 ms intervals, so bounces should be settled by this time. There are no methods to check if the queue is full prior to inserting an event. Thus, unprocessed events might be overwritten by incoming events. This is part of the problem analysis process. The application developer must understand the worst case queue usage, and provide for efficient error handling if queue size exceeded.

Conclusions

In this exercise, we have implemented the user-written code that is required for a visualSTATE application. We have created actions that are executed at periodic intervals, implemented concurrent behavior, and implemented guard expressions.

References

1 Melkonian, Michael: “Getting by Without an RTOS”, *Embedded Systems Programming*, September 2000. This is a very readable discussion of using queues and timer objects to implement desired behavior in embedded applications. All you need to add is visualSTATE to handle your control logic.

2 Oshana, Rob: “Sequence Enumeration”**,** *Embedded Systems Programming*, September 2000. This article discusses a concrete procedure to analyzing a problem and coming up with a statechart diagram.

3 Horrocks, Ian: *Constructing the User Interface With Statecharts*, Addison Wesley, 1999. This short book describes the use of statecharts to model applications, and gives three simple and practical case studies.

4 *IAR visualSTATE Quick Start Tutorial*. This is a step-by-step tutorial which demonstrates how you design, test, implement and complete an embedded application by means of visualSTATE.

5 *IAR visualSTATE Reference Guide* describes the constructs, elements and principles of state machine modeling that are available in visualSTATE. For example it gives detailed descriptions of constructs such as *states*, *transitions*, *state reactions*, *action functions*, *guard expressions*, *visualSTATE Projects*, etc.

6 *IAR visualSTATE User Guide*.

7 *IAR visualSTATE API Guide*.

Contact information

SWEDEN: IAR Systems AB
P.O. Box 23051, S-750 23 Uppsala
Tel: +46 18 16 78 00 / Fax: +46 18 16 78 38
Email: info@iar.se

USA: IAR Systems US HQ - West Coast
One Maritime Plaza, San Francisco, CA 94111
Tel: +1 415 765-5500 / Fax: +1 415 765-5503
Email: info@iar.com

USA: IAR Systems - East Coast
2 Mount Royal, Marlborough, MA 01752
Tel: +1 508 485-2692 / Fax: +1 508 485-9126
Email: info@iar.com

JAPAN: IAR Systems K.K.
1-2 Kanda-Ogawamachi, Chiyoda-ku
Tokyo, 101-0052
Tel: +81 (0)3 3251 0886 / Fax: +81 (0)3 3256 4791
Email: info@iarsys.co.jp

UK: IAR Systems Ltd
9 Spice Court, Ivory Square, London SW11 3UE
Tel: +44 20 7924 3334 / Fax: +44 20 7924 5341
Email: info@iarsys.co.uk

GERMANY: IAR Systems AG
Posthalterring 5, D-85599 Parsdorf
Tel: +49 89 900 690 80 / Fax: +49 89 900 690 81
Email: info@iar.de

DENMARK: IAR Systems A/S
Lykkesholms Allé 100, DK-8260 Viby J
Tel: +45 87 34 11 00 / Fax: +87 34 11 90
Email: info@iar.dk

September 2001.