Real-Time Data Logger

- Many applications rely on the timely acquisition of sensory data; the samples have to be taken at a pre-defined sample rate and uploaded to a host computer
- Simple real-time programs rely on timer interrupts to ensure that all real-time tasks run in appropriately scheduled time slots; more complex timing issues are commonly solved using real-time operating systems
- A simple real-time data logger can be implemented by reading out the ADC result register from within a timer interrupt service routine; the thus acquired data can be stored or directly uploaded to the host

<table>
<thead>
<tr>
<th>week</th>
<th>lecture</th>
<th>topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Embedded Control Applications III</td>
<td>- Real-time data logger (menu driven, serial communications, adjustable sample rate)</td>
</tr>
</tbody>
</table>

Real-Time Data Logger

- This lecture will outline the design of such a simple real-time data logger for the C167 microcontroller
- Timer interrupt T0 is used as timing engine; this is one of the fast Capture-Compare timers (maximum resolution: 8 CPU clock cycles → 8/20⋅10^6 = 400 ns)
- Sample periods from 100 µs to 3.36 s can be chosen – to extend this range, timer T0 would have to be cascaded with another timer (e.g. GPT timer T6)
- The T0 Interrupt Service Routine (ISR) is used to read out the ADC unit (channel 0); this data is time stamped and stored in a circular buffer

- Upon acquisition of the requested number of samples, the entire data is uploaded to a host computer via the serial interface ASC0; opening a terminal program (56700 bps, 8/N-1) allows the data to be monitored in tabular format
- As the program will be extended to acquire data straight into the MATLAB workspace, all clear text messages can be switched off using a global macro MATLAB; calling the compiler with this macro defined, causes it to ignore all those code sections which would produce these messages
Real-Time Data Logger

- The serial communication interface is implemented using blocking functions (polling); future extensions could replace these functions by the background communication interface developed in lecture MP7.
- A modular approach is taken to keep functionally distinct functions in separate source code files; this will greatly facilitate the maintenance and future extensions of the program.
- The timing of the real-time engine can be monitored on port 2 pin 0 (P2.0); definition of macro TIMINIG causes the compiler to include the relevant code.

Real-Time Data Logger: Main program

- This is the main module; all global variables are defined here (and exported via the DAC167.h)

```c
#include <reg167.h>                      /* special function register C167 */
#include <stdio.h>                       /* sscanf() */
#include "DAC167.h"                      /* global project definition file */
#include "serial_poll.h"                 /* serial communication routines */
#include "timer.h"                       /* timer functions */
#define TIMINIG /*/                      /* global variables (system wide) */
unsigned int stillToRead; /* number of samples still to be read */
unsigned int savindex; /* current 'save index' [save_record] */
unsigned int savfirst; /* first 'save index' [circular buffer] */
struct mrec save_record[SCNT]; /* buffer for measurements */
unsigned long int usSamInt; /* sample interval in microseconds */
```

Real-Time Data Logger: Main program

- This is the main module; all global variables are defined here (and exported via the DAC167.h)

There are truly global variables (visible to the other modules) and static global variables (only visible within the current source code file).

Variable startup_message[] has been defined using the storage class specifier const; this causes the associated text to be kept in ROM only.

Global variable save_record and current are of type struct mrec; this structure has been defined in header file DAC167.h.
Real-Time Data Logger: Main program

```c
void main(void) {
    /* toggle LED - allows timing to be verified on P2.1 */
    #ifdef TIMING
        DP2 |= 0x0002; /* define P2.1 as o/p */
        P2  |= 0x0002; /* set P2.1 high (inverse logic) */
    #endif
    serial_init(); /* initialize ASC0, 57600 bps, 8/N-1 */
    IEN = 1; /* allow all enabled interrupts to be triggered */
    ...
```

Macro TIMING can be set to measure the timing on P2.1: the pin is set low at the beginning of the ISR and reset to high at its end.

Upon initializing the serial interface, all enabled interrupts are allowed

Real-Time Data Logger: Support functions

- The support functions keep the code readable:

```c
void init_vars(void) {
    int idx; /* index for circular buffer */
    /* reset clock record */
    current.time.hour = 0;
    current.time.min = 0;
    current.time.sec = 0;
    current.time.msec = 0;
    current.time.usec = 0;
    /* initialise circular buffer */
    sindex = savefirst = 0; /* reset circular buffer index */
    for(idx = 0; idx < SCNT; idx++) /* mark all records as 'unused' */
        save_record[idx].time.hour = 0xff; /* (unused flag: hour = 0xff) */
    ...
```

A slightly wasteful way of keeping track of the time...

Real-Time Data Logger: Support functions

Support function menu reads the number of samples as well as the sample rate from the command line (terminal, connected on ASC0)
Real-Time Data Logger: **Support functions**

```c
#ifndef MATLAB
else myPrintf("Acquiring %d samples\n", nSample);
#endif /* set number of samples that still have to be read (all of 'em) */
stillToRead = nSample; /* get sample period in micro-seconds */
#ifndef MATLAB
myPrintf ("Enter sample time (microseconds, min. 100): ");
#endif getline(&cmdbuf[0], sizeof (cmdbuf));       /* read command line (with echo) */
for (i = 0; cmdbuf[i] == ' '; i++) { ; }    /* skip leading blanks */
sscanf(cmdbuf, "%ld", &usSamInt);           /* scan input for sample rate */
if(usSamInt < 100) {
#ifndef MATLAB
myPrintf("Too fast. Increasing to 100 microseconds (0.1 ms)\n");
#else /* MATLAB */
usSamInt = 100;                           /* micro-seconds */
#endif
}
#ifndef MATLAB else myPrintf("Sampling interval: %ld microseconds\n", usSamInt);
#endif /* menu */
```

Support function `wait_for_start` calls upon `_getkey()` to receive a character on serial reception line RxD; the function `blocks` as long as this character (once received) is not equal to 's' – this is a simple way of implementing one-way handshaking.

```c
/* display current contents of measurement buffer */
void display_data(struct mrec display) {
    unsigned char i;                            /* index count for AN0 - AN3         */
    #ifndef MATLAB
    myPrintf("%2d:%02d:%02d.%03d%03d  ",
            display.time.hour, display.time.min,  /* display hours and minutes         */
            display.time.sec, display.time.msec,  /* display seconds and milliseconds  */
            display.time.usec);                   /* display microseconds              */
    for (i = 0; i < nCHAN; i++)               /* display AN0 through ANnCHAN */
        myPrintf(" AN%d : %4.2f V", i, (float)(display.analog[i]) * 5.0 / 1024);
    #else /* MATLAB */
    for (i = 0; i < nCHAN; i++) {
        myPrintf("%d
", display.analog[i]);    /* display AN0 through ANnCHAN (raw) */
        blinky(1000);                           /* visual feedback during upload */
    }
    #endif /* MATLAB */
}
```

Support function `display_data` displays the contents of a single data record; in **terminal mode**, each record includes a time stamp and the formatted data of the ADC channel(s); in **MATLAB mode**, only raw data values are sent and data the upload is signalled using the LED.

```c
int idx;                                       /* index for circular buffer */
#ifndef MATLAB
myPrintf("Time: %2d:%02d:%02d.%03d%03d  
",
    display.time.hour, display.time.min,  /* display hours and minutes         */
    display.time.sec, display.time.msec,  /* display seconds and milliseconds  */
    display.time.usec);                   /* display microseconds              */
for (i = 0; i < nCHAN; i++)               /* display AN0 through ANnCHAN         */
    myPrintf(" AN%d : %4.2f V", i, (float)(display.analog[i]) * 5.0 / 1024);
#else /* MATLAB */
for (i = 0; i < nCHAN; i++) {
    myPrintf("%d
", display.analog[i]);    /* display AN0 through ANnCHAN (raw) */
    if (++idx == SCNT) idx = 0;             /* next circular buffer entry  */
}
```

Support function `upload_data` calls upon `display_data` to send formatted (or raw – **MATLAB mode**) data records to the host; the function simply loops through all of the currently stored records – memory `save_record` is made circular by resetting the index variable `idx` to '0' when moving beyond the end of the buffer.
Real-Time Data Logger: *Timing*

- The heart of the data logger is the timer ISR:

```c
static void timer0_ISR(void) interrupt 0x20 using INTREGS {
    unsigned int i;
    #ifdef TIMING
    P2 ^= 0x0002;   // clear bit on entry (inverse output logic)
    #endif
    /* update current time */
    current.time.usec += usSamInt;
    ...
}
```

Definition of macro TIMING allows precise timing information to be produced: upon entry to the ISR, bit 1 of port 2 (P2.1) is set *low*; this bit is reset to *high* before exiting from the ISR.

The micro-second counter of variable *current* is increased by the per-interrupt increment *usSamInt* and all other counters are adjusted, if required;

Upon having acquired all selected channels, the *current* measurement is stored in the ring buffer and all index variables are updated; if the store index (*sindex*) exceeds the buffer maximum (*SCNT*) it is reset to *0*.

Once the requested number of samples have been taken (variable *stillToRead* is *0*) timer T0 stops itself.

Real-Time Data Logger: *Timing*

- The period of the timer is set by *init_T0*:

```c
/* initialize timer T0 */
void init_T0(void) {
    unsigned long int period; /* determine prescale value (T0I) */
    unsigned int T0I; unsigned int RLvalue; /* reload value */
    /* calculate reload value */
    period = 26250; /* prescaler 8 (000) */
    T0I = 0; /* 000 */
    /* determine suitable period */
    while((usSamInt > period) { /* corresponding TOI */
        ...
    }
    ...
}
```

Function *init_T0* first determines the required pre-scale value (*T0I*) by successively doubling the period, beginning with its minimum value 26250 µs (26.25 ms) until *period* exceeds the requested sample interval.
Real-Time Data Logger: Timing

```c
/* set reload value and program T0I */
RLvalue = (unsigned int)(0xFFFF*(1 - (float)(usSamInt)/period));

T0REL = RLvalue;
T0    = RLvalue;
T01CON |= T0I;  /* setup the timer T0 interrupt */
T0IC  = 0x44;   /* set T0IE and ILVL = 1, GLVL = 0 */
```

Function `init_T0` then programs the reload value (`RLvalue`) and the interrupt level for timer T0 (ILVL : 1, GLVL : 0).

Function `start_T0` has been provided to start the timer from within higher level functions (main).

Real-Time Data Logger: Serial interface

- The serial communication routines have been extended by `getline` and `myPrintf`:

```c
#define CNTLQ 0x11
#define CNTLS 0x13
#define DEL    0x7F
#define BACKSPACE 0x08
#define CR      0x0D
#define LF      0x0A

void getline (char near *line, unsigned char n) {  
  unsigned char cnt = 0; char c;
  do  {
    if ((c = _getkey ()) == CR)  c = LF;    /* read character                 */
    /* input character */
  } while (cnt < n - 1  &&  c != LF);      /* check limit and line feed      */

  *line = 0;                                /* mark end of string             */
}
```

Function `getline` reads from the serial interface ASC0 until a carriage return (CR) character is received; this character is replaced by a line feed (LF) to mark the end of the string.

```c
#define STRING_BUF 300                       /* size of local buffer for strings */
static char userbuf[STRING_BUF];             /* static -> local to this file */

void myPrintf(char *fmt, ...)
{
  va_list arg_ptr;
  va_start(arg_ptr, fmt);                    /* fmt : format string */
  vsprintf(userbuf, fmt, arg_ptr);           /* vsprintf accepts ptr to args... */
  va_end(arg_ptr);
  putline(userbuf);
}
```

Function `myPrintf` simply extracts the always present format string (`fmt`) and a pointer to an optional list of further parameters (`arg_ptr`) before calling `vprintf`, the version of `sprintf` which accepts a variable number of call-up parameters.

This allows for calls to `myPrintf` with and without a variable parameters: `myPrintf("hello\n");` and `myPrintf("%d\n", myInt);`
Real-Time Data Logger

Simulation allows validation of the code; the timing, however, can only be checked on the actual hardware.

- Upon download to the Flash EEPROM of the target, the code can be controlled using a terminal program.

- The requested timing can be checked by checking the voltage on port 2, pin 0 with an oscilloscope.

- The display of timing information on P2.0 can be enabled using compiler macro `TIMING`.

- Example: Acquisition of 100 values from ADC channel 0, sample period: 100000 μs = 100 ms

- T0_ISR is called every 100 ms (4 divs at 25 ms / div)

- Duration of the recording: 100-100 ms = 10 seconds
Real-Time Data Logger: *MATLAB interface*

- With minor modifications, the data logger program can be used from within MATLAB – this way data can be *read directly into the MATLAB workspace* where it can be analysed and/or processed.
- In its present form, the logger first acquires all requested samples and then uploads them to the host; an improvement would be to use the *background communication routines* developed in lecture MP7 – this way, ‘live’ data could be visualized.
- Here, we shall restrict ourselves to the *sequential* version, i.e. the upload *follows* the acquisition phase.

The data logger on the C167 initially expects the number of samples (*nSamples*) to be sent, followed by the sample period in microseconds (*period*); the data logger echoes all received characters back to the host – the m-file therefore has to read/remove this data from the reception buffer, even if this information is not used (dummy call to fscanf).

Data acquisition is initiated by sending character ‘s’.

MATLAB has built-in commands which *open/administer/close* a serial port object; command *serial* accepts a number of parameters which define the characteristics of the port.

To *open*, *write to*, *read from* and *close* a serial port, MATLAB provides commands *fopen*, *fwrite*, *fprintf*, *fread* and *fscanf* and *fclose*, respectively.
Real-Time Data Logger: *MATLAB interface*

```matlab
(...) % receive data values
myData = zeros(nSamples, 1);
for(i = 1:nSamples)
    myVal = fscanf(sp_ID, '%s');
    myData(i) = str2num(myVal)/1023*5;
end

% plot results
plot(t, myData, 'g.-')
title('Data, channel 0 [0 ... 5 Volt]'); xlabel('time (ms)'); ylabel('Vin [V]');
end
```

On completion of the data acquisition phase, the logger uploads all data values in its raw 2-byte format; the *values arrive as 'text' – the m-file therefore has to convert from strings to numbers*. The re-scaled data values are displayed using the *'plot' command*.

---

**Embedded Control Applications – Outlook**

- The presented program is just a *quick'n'dirty* example of a host-target interface; more serious applications should *consider transmission errors* and implement a *protocol with handshaking*.

- Transmission errors may lead to situations in which either of the two communication partners expects a signal which will never arrive; this pitfall can be avoided with the use of *timeout timers* (on both sides!) – the design of a robust communication system is not straight forward and is best done using *state diagrams*.

- *State diagrams* help a software designer to visualize the sequence a program runs through in the presence and/or absence of *internal and external signals*.

- Example: *(generic microcontroller program)*

![State Diagram Example](attachment:image.png)
Embedded Control Applications – Outlook

- The design of a *communication system* often makes use of *telegram sequence diagrams* to visualize the flow of telegrams between the individual partners.

Send command C1  
Timeout timer…  
response must have been received before the timer elapses  

Acknowledge command C1  
Timeout timer…  
a telegram must have been received before the timer elapses  

Next data telegram (solid) or fault indicator telegram (dashed)

- Microcontroller based software almost always benefits from a clear and modular structure; the use of state diagrams greatly assists the programmer in developing such a structure.

- The ever increasing complexity of embedded systems has led to a number of formal approaches to software development, e.g. the *Universal Modelling Language (UML)* and corresponding software engineering tools.

- *Embedded software engineering* has become so complex that entire degrees focus on nothing else…