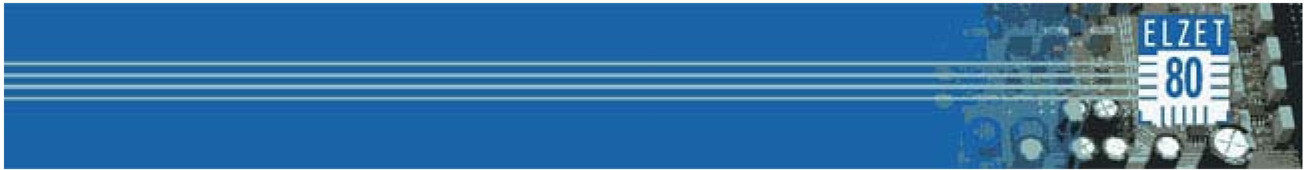
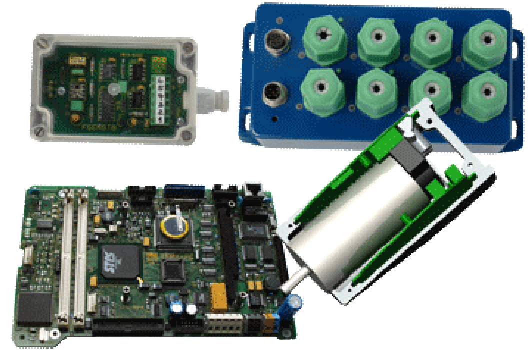


ELZET80



mCAT -
A real time kernel for TLCS900 and ARM processors

Why does ELZET80 strongly recommend to use the mCAT2 real-time kernel for programming?

Here´s a list of the benefits:

Multiple Tasks

Independent programs can concurrently serve different tasks like communication with a PC, pressure regulation, or local keyboard handling.

Modularity

If it´s easy to split programs into different tasks, the programmer tends to isolate certain functions to a dedicated task. It can be debugged and tested on its own. Very probably some of these tasks are not confined to one automation problem but may be re-used next time.

Interrupt drivers

Events can be handled by user written (C-) code in an orderly and consistent manner. They can be re-used if the task changes.

Function libraries

mCATs modular library system "ShLib" starts small with the marginal kernel functions. Libs can be added as needed and system as well as user libs use the same calling mechanism (SWI).

Support programs

A monitor displays memory and task/int status. It serves as the receiving end for program downloads into RAM or Flash- EPROM. A ticker times the operation of tasks while BITBUS support opens the target to the fieldbus- network. User-I/O makes hardware- I/O easily accessible.

Real time executives have been around for microprocessors since the late 70's and all have a similar goal: To allow the different concurring tasks a processor has to solve to cooperate in an orderly and serviceable manner.

mCAT is a product of the 90's and builds on new ideas like distributed processing, object orientated

Order Codes:

200.- € *

SFT-MCAT2*

(Version 2.x) Real time executive, developers pack. Tools, samples and a printed manual. For use with nearly all ELZET80 hardware.

0.- € *

SFT-MCATRUN

Runtime licence for 2+ mCAT is part of the hardware for all newer products.

from 20.- € *

HDB-MCAT2

mCAT2 printed manual.

* All prices in EUR ex Aachen, plus taxes where applicable
* Software prices do not include EPROMs or Flash-EPROMs.

programming and open communication to name a few. mCAT has been designed as a real time executive for a (BITBUS) fieldbus environment for both the master and the slave processors. Distributed control architectures require that processes c ommunicate by sending messages to each other as other mechanisms like global variables or flag arrays are restricted to one CPU only.

Although forced by the requirements of distributed control, message passing as inter- process-communication (IPC) has many advantages over other approaches, the most important being truly modular design with really reuseable modules. Every task can be split into client and server processes and - usually automatically by the nature of the concept - the server task tends to be universally useable not only for the project just under your fingers. Imagine a server task to support filtered analog input: it needs a startup message to set the channels you are interested in, to select the scanning interval and the type and depth of the filter. After this, your client task (a regulator or level control unit) simply asks for values and gets a message with the channel values selected. With the next project you might need to include linearization or alarm level monitoring into the analog input function. Then the task could be extended to support linearized values alternatively - which is easily done as the task is isolated and has known inputs and outputs.

Further the viewpoints of client and server can be flipped or even mixed with small modifications: have the analog input task check the alarm levels and then send a message to an alarm handler task if the values are high. This puts the task into a client status for this part of the job (the alarm task is the server as it processes the alarm protocol on behalf of the analog input task and replies if it's ready). Still, at the same time the analog input task continues to be the server for those tasks needing filtered values on demand. It will also be rather simple to port the application to a hardware with another ADC - the analog input task would have to be modified but the rest of the application could stay untouched.

Priority messaging

Unlike other real time executives mCAT works with dynamically changing task priorities. There is a start priority defined in the task header but in real use the task inherits its priority from the arriving message. The idea behind the concept of passing priority with a message is that in most cases the task producing information knows best how urgently it must be handled by the consumer. Take our example above: the analog input task can send a high priority message if a value crosses the alarm border while a low priority is sufficient to print out the mean value over the last five minutes.

Traditional kernels handle these exceptions by using an extra task with high priority that sleeps for most of the time. If an alarm condition occurs, this task is signalled and starts working. The mCAT approach is a lot simpler: one consuming task can handle all exceptions as it is put to high priority by the urgent message that arrives and switches back to standard priority after this message has been processed.

Under mCAT task priority is always equal to the priority of the highest priority message waiting at one of its queues.

EiIn the case a task is actually running (i.e has the processor) at a low priority and a high priority message arrives it is immediately put to the high priority and continues execution. This is the fastest way possible to have the task execute the newly arrived message. As messages are sorted by priority, an urgent one doesn't have to

wait until older messages are processed but immediately gets through. Messages at the same priority are sorted by age.

Tasks

To start with the simple: A program in its minimal form consists of just one task with the default main thread and one queue for incoming messages.

A task has a minimum of three parts:

task header IMD (Initial module descriptor)

INIT function

MAIN function

and there's a default QUEUE assigned to each task.

Task header IMD

mCAT scans the EPROM at cold start to configure itself. It searches for Initial Module Descriptors (IMD's) that all begin with the same bit pattern (AA55 Hex). The table that follows this marker contains information about the version number of the module, its initial priority, stack size etc. plus pointers to the INIT and MAIN code area. An MS-DOS utility program generates the IMD structure.

INIT function

After mCAT has found an IMD and checked the validity of the check sum, it calls the address found in the table entry for the TaskInit() function. This serves to prepare a tasks environment, if necessary. A task that handles the display for example, might initialize the display controller and clear the screen.

MAIN function

As soon as the TaskStartup has been executed, mCAT includes the task in its list of active tasks (tasks ready to run). If it has the currently highest priority, mCAT will call the TaskMain() function. This may be immediately or never or anything in between, depending entirely on the priority of other tasks. After the task has acquired the processor, it will not loose it before some other task gets a higher priority - which can only happen by an interrupt (serial line, timer) or by the task itself sending a message with a higher priority to another task.

The other way to loose it is to give it up by calling a WAIT function. Usually, a task will need some input data to do something meaningful, so it will have to tell another task or an interrupt driver to provide it with messages. After this, the task has to check for such incoming messages. While it could loop around a MsgGet function call and keep the processor, this would unnecessarily consume processor time if there's no message waiting and prevent lower priority tasks from running. The best way to wait for a message is the MsgWait function call: it will put the task itself into the sleep mode until a message is received. It then is woken up at the priority level of the incoming message.

After executing whatever has to be done with the incoming message, the task usually loops to the MsgWait function to wait for the next message. The call to TaskMain() never really returns.

Messages

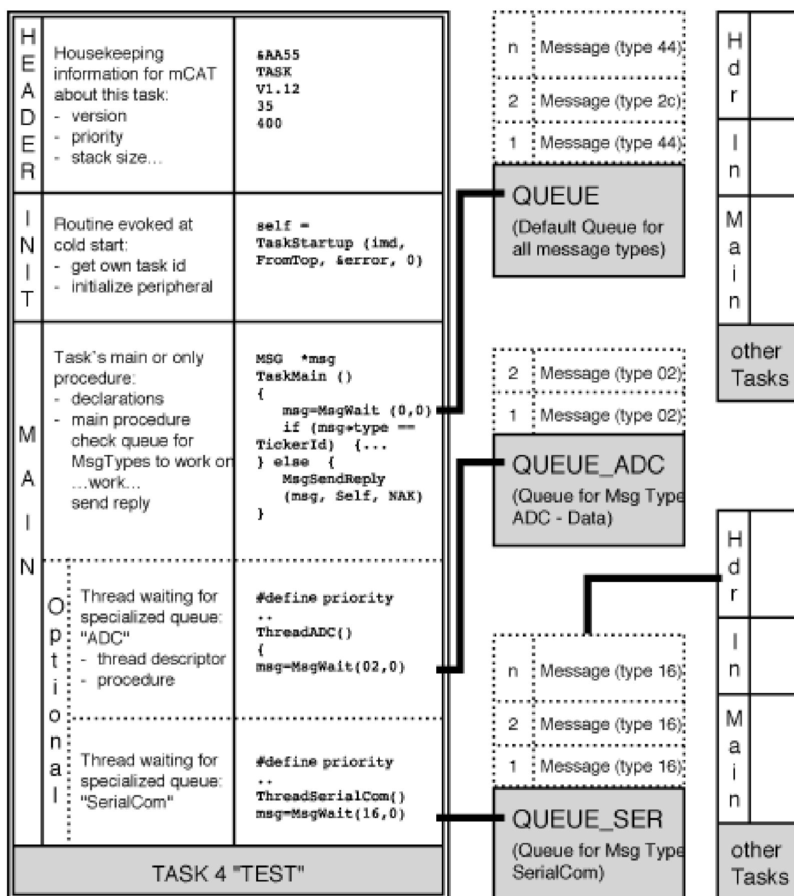
A message is a data structure with a descriptive header and the actual data as message body. The header is a table with information about the source task, the priority and the reply priority, the length and a message type identifier MSGID that is

used to differentiate between groups of messages. Messages can be sent by tasks or interrupt drivers and can be sent to tasks only.

Message types

mCAT imposes no restriction on the data content of a message as long as it fits into memory, a length less than a kilobyte is recommended however. To ease message handling, mCAT suggests however to use groups or types of messages. It is advised that the server task defines a message type (for example a message for printer output or one for analog data) because the server is the one to receive this message and work on it.

To create a message type the server defines a name (ASCII characters) and calls the MsgIdCreate function to assign an ID to the messages of this type. The ID assigned by mCAT to a message type is called msgid and is for example used in queues to obtain an order of messages sorted by type. Other tasks, like necessarily the client task, can use the MsgIdQuery function to find out the ID code of a message type with a given ASCII name.



Structure of an MCAT task and its queues

Queues

Each task has at least one queue to receive incoming messages, if it has more than one, all but the default queue are for messages of one type only (they have the same MSGID). With each call to GetMsg or WaitMsg the oldest msg waiting at the queue is retrieved. In fact, no data is passed when a message is transferred but simply the pointer to the message. It's the responsibility of the programmer to make sure the sender does not overwrite the message before the receiver has been working on it. mCAT supports the synchronization by the reply mechanism. The sender may give the message a reply priority. The receiver copies this into

the priority field of the message and clears the reply field to zero. This is however not left to the user, the MsgSendReply function handles this task. A call to the MsgSendReply function with a parameter of "ACK" or "NAK " (i.e. acknowledge or not) returns the message to the sender. You may send a message with a reply priority value of zero if you are not interested in replys.

Ticker

For a typical regulation or data collection application you need to get a task done in regular intervals. In mCAT there's a special interrupt driver that provides timing for such applications. The so-called ticker can be set up to send a message every n milliseconds to a task at the requested priority. According to the general working scheme of mCAT, this task is put to the priority of the ticker message immediately at receipt. The called task replies to the ticker after processing the timer message. Granulation of the time interval depends on the available hardware, usually ticker messages can be set to multiples of 10ms.

The function ALL sets up the ticker like ALL(&tick,5000l, 100,Self). This will result in the ticker message tick to be sent every 5000 milliseconds to my task (Self) at priority 100. The ticker message tick is an instance of the predefined TickerMsg and needs no user modification. On the following pages, the Ticktest sample program will show how to write a task for mCAT.

Example: Ticktest

To get a first idea about how the actual code looks like, here's an example of a simple program that writes out a counter every second to the terminal serial line. To accomplish this, it will request ticker messages every second, increment a counter variable and print it. Program text starts including the necessary header files. MCAT.H is the main include file, itself including headers like IMD.H, MSG.H etc., where the basic data structures of mCAT are defined. TICKER.H provides the ticker message structure and the setup function definitions for ALL and AFTER while IO.H defines basic serial line i/o-functions like printing a string (WrStr) or a long word in hex notation (WrHexLWord).

```
/* '''''' {MODUL ticktest} '''' ''''''''''''''''''''''''''''''''''''
    (c) 1995 Volker Goller, Aachen
    Project :           mCAT V2.0
    Function:           TickerDemo
    History :           VER    |DATE            |WHO?    |WHAT?
                          '''''' '''''''''''''''''''''' '''''
                        001    | 09/20/95       |VG      |initial
*/
#define __MOD_TICKTEST

/* THIS FILE CAN BE COMPILED WITH TOSHIBA C V1.03 AND V4.0   */

#include <mcat.h>              /* mCAT functions and datatypes   */
#include <ticker.h>            /* ticker msg and functions */
#include <io.h>                /* Simple, non standard io functions
                               With version 4.0 of the Toshiba C-compiler
                               you may even use "printf" */
/*  ''''''' '''''''''''''''''''''''''''''''''''''''''''''''''''''''''  */
/*                      INCLUDE IMD FILE (task header)              */
/*  ''''''' '''''''''''''''''''''''''''''''''''''''''''''''''''''''''  */
#include "ticktest.imd"

/*  ''''''' '''''''''''''''''''''''''''''''''''''''''''''''''''''''''  */
/*                      INIT TASK                                   */
/*  ''''''' '''''''''''''''''''''''''''''''''''''''''''''''''''''''''  */
void TaskInit (imd)     /* new with version 2: own IMD as an argument */
IMD *imd;
{
    int error;          /* you may or may not check for errors
                        There must be a fundamental problem if
                        the task will not start */

    Protect();          /* Neccessary to avoid problems while
                        init the task from within the monitor */

    Self = TaskStartup(imd,FromTop,&error,0);
                        /* imd      = pointer to own imd
                           FromTop  = alloc tasknumber starting from
                                      the highest available down to 0
                           error    = pointer to a error variable
                           0        = reserve no space for extra
                                      queues. The default queue is
                                      not affected by this statement. */
UnProtect();            /* re-allow task switching */
}
```

```
/*  ...... ...........................................................  */
/*        DECLARE YOUR VARIABLES HERE                                    */
/*  ...... ...........................................................  */

PRIVAT TickerMsg tick; /* used to request service by the ticker */
PRIVAT MSG *msg;       /* a pointer to handle incoming messages */

void TaskMain ()
{
    lword pock;                    /* a "tick counter" */
    pock = 0l;                     /* clear tick counter */
    ALL(&tick,1000l,200,Self);     /* request the ticker to send "tick"
                                      all 1000ms (1sec) to myself (Self)
                                      using a priority of 200 */
    WrStr(" *** TICKTEST 1.0 ***\n\n\r");
                                   /* hello */

/*  ...... ...........................................................  */
    /* Some debuging code. I left it to show some internals */
    WrStr("Self : ");WrHexWord(Self);WrLn();
    WrStr("TICK ID: ");WrHexLWord(TickerId);WrLn();
/*  ...... ...........................................................  */
    /* Ensure that you NEVER return from this TaskMain function! */
    /* The only way to exit is to call "TaskDelete(Self);"       */
/*  ...... ...........................................................  */


   while (1) {
        msg = MsgWait(0,0);        /* wait for a msg. Timeout is NULL,
                                      use default queue "0" */
        /* we got a msg. Check if it was a ticker msg */
   if (msg->type == TickerId){
                    /*  .................  */
                    /* IT'S A TICKER MSG   */
                    /*  .................  */
                    MsgSendReply(&tick,Self,ACK);
                    WrChar('\r');          /* output tick counter */
                    WrDecLWord(++pock);
                    WrStr(" sec since start");
       } else {
                    /*  .................  */
                    /* UNKNOWN MSG!        */
                    /*  .................  */
                    MsgSendReply(msg,Self,NAK);
                    WrStr(" *** FAIL ***\n");
           }
       }
}

/*  ```  (END OF MODUL ticktest)/*  ........ ..... ................  */
```

IMD

Next the task header IMD is included. The imd will be generated by the utility IMD.EXE and takes ".cm" as extension. If you want to modify startup priority, <<ticktest.cm>> must be edited.

INIT

The INIT part of the task - TaskInit(imd) - follows with defining an instance of the IMD data structure. An error variable is declared - although not used in the program. Then, a call to Protect() locks task switching for the following critical section. TaskStartup creates a task, assigns it a task number (returned into Self) and activates it immediately at the priority given in the IMD. It will retain this priority until it gets its first message, whereafter it is put to the priority of the message. The last parameter of TaskStartup() is set to 0 as we don't want more than the default message queue. The call to UnProtect() should be done immediately after leaving a critical section, so the task switching can continue. This ends the TaskInit. If Ticktest is the only task, it will now continue in the TaskMain() function, if there are other tasks, it depends on the

priority of the other tasks, which one is going to get the processor. Anyway, sooner or later mCAT will call TaskMain() to execute.

main

TaskMain() is the main part of the task, it would be main() in a standard C environment. First an instance tick of the ticker message is created and a pointer to the incoming messages declared. pock is going to be the counter variable that we will print out as the number of messages received.

The function ALL is parameterized to use the message tick to be sent all 1000 milliseconds at a priority of 200 to my task. A greeting message is printed and the following debug section lets the user check for the assigned task number and ticker id. The program continues as an endless loop. At the begin of the loop, you notice the probably most often used mCAT call: MsgWait (queue_handle,timeout)

MsgWait tells mCAT to wait for an incoming message on queue 0 in our example, which is the default queue that gets all messages that don't have a special queue opened. A timeout of 0 indicates that it will wait forever if no message arrives, a long value greater zero would define a timeout in milliseconds. The function would return -1 in case of a timeout.

Now the program has to check the type of the message. As we are waiting for a message from ticker, the type field in the msg data structure (msg->type) must contain the value TickerID. If this is correct, the ticker has to be notified that the message arrived ok. mCAT provides the function MsgSendReply for this. It returns the message with a handshake value of ACK or NAK.

If the arriving message was a message from ticker, the MsgSendReply is returned with acknowledge, pock is incremented and written to the terminal serial line. If the message is unknown, it will be refused with the reply value NAK. After this, a new MsgWait is issued in the loop.

This is a complete task under mCAT and may serve to extend it to an analog input scanning program or a regulator task for your own application. This and other sample sources are part of the mCAT2 developers diskette.

Kernel extensions

mCAT consists of three major program types:

Shared libraries
Interrupt drivers
Tasks

As already mentioned, modularisation for reusability has been one of the main goals for mCAT. The Shared Library supports this concept by defining a mechanism for a flexible arrangement of system components. The kernel itself is a shared library.

To be useful for more than just the most basic tasks, the kernel should be extended by other programs to ease programming for more demanding tasks. There are three extensions required for minimum system configuration (as for example even the kernel uses them):

the memory manager
the name server and the
non-volatile memory manager

Memory Manager

The memory manager allocates memory, which usually is done implicitly using the standard functions. There is another memory manager especially addressing non-volatile memory like the serial EEPROM on most ELZET 80 hardware and FLASH EPROM. Functions are available to determine the type of memory available and to perform basic read and write.

The Name Server program is a directory program that maintains the ID's of tasks and interrupt drivers. It returns the ID if the ASCII name is presented to the name server using the MsgIdQuery system function. This mechanism allows a uniform access to tasks regardless of the system configuration just by knowing their name. Also, for different purposes, tasks with different complexity but with the same name can be used, which again supports reusability.

Useful extensions: Ticker, Sysmon.

Usually, your mCAT configuration contains the Ticker interrupt driver. It provides timing and scheduling services for your tasks. The system function AFTER sends a message to the task that called it after the time specified in the function call has elapsed. ALL extends the functionality of AFTER by repeating the message over and over again with the same time interval. The Ticker waits, however, until its message is confirmed by ACK (want more messages) or NAK (stop sending messages), before sending new messages. If your tasks answer is delayed and a new message has been due before the acknowledgement, the ErrTicksLost code can be found in the MSG structure (mymsg.msg.error). A sample program using the ticker has been described in the introduction of this manual.

Another useful extension in mCAT is the monitor task. It displays memory and allows modifications, gives status information about tasks and interrupt drivers, loads programs over the serial line and manages manual task creation and deletion.

BITBUS

BITBUS support is one of the extensions not everybody needs with mCAT, although the majority of applications meanwhile provides fieldbus support. It takes two programs to run BITBUS: the interrupt driver BitbusDrv and the GBS (RAC) task. The interrupt driver handles the layer 2 services of BITBUS which means it provides all functionality up to the message level: IUSC network controller communication using interrupts and DMAs, message buffering and low level control message handling.

BitbusDrv sends a finished message to the consuming task and accepts replies for transmission over the network. GBS (Generic Bus Services) is the IEEE1118 successor of the RAC-task as it was known under BITBUS. GBS provides a basic functionality common to all BITBUS stations. It includes reading and modifying i/o and memory, up- and downloading code, creating and suspending tasks, time services and many more. Every mCAT slave supports the basic conformance of IEEE1118 GBS plus many of the extended functions. GBS always has to be task 0 in mCAT to be compatible with BITBUS requirements, its standard priority is 240. The address extension of GBS/RAC-commands beyond the standard BITBUS 64K memory is implemented according to IEEE1118 recommendations, allowing access to all TLCS900 memory and resources.

Waiting for a BITBUS message.

Working with BITBUS requires straightforward message handling as with all other mCAT operations. The ID of the driver is obtained by calling MsgIdQuery with the globally defined constant BitbusName.

Then the standard MsgWait system function is called and if the function returns, the MsgId is compared to the Bitbus ID. Now your task works on the received BITBUS message and returns it to the master by using the standard MsgSendReply function.

Add-ons: Express-I/O, SerDrv, BgMem, UserIO..

mCAT makes it easy to add modules to the system, hence ELZET80 continually extends the functionality to make programming faster and easier. The extension with the most time saving effect is Express-I/O - giving access to analog and digital inputs and outputs in a high-level manner. I/O-objects can be defined for all relevant ports with an evident name so they can be accessed with i/o-macros like out(&belt_speed, 176). In addition, there are system supported background functions that add edge detect, counter or pulse generation capabilities to ordinary digital i/o.

For the moment, the serial drivers supplied with mCAT for the TLCS900's own two UARTs are polled drivers. Buffered interrupt drivers (called SerDrv) are available separately. They allow transfer of data between the master and the two serial ports with no mCAT coding on the slave side. There is a separate documentation available for SerDrv.

BgMem is a background storage for equal length "records" like sets of data values from measuring inputs, alert and event logs, configuration data etc. Storage options are FIFO, LIFO or ring buffer to adapt to about all different logging needs or random for more data-base-like operations. Records and the whole "file" are safeguarded with CRC generation and checking; if a real-time clock is available, file creation and modification times are available. BgMem needs hardware with battery backed-up RAM

For hardware containing keyboard and LCD character displays an mCAT task called UserIO is available. It scans the keyboard, beeps the beeper and puts messages to the LCD (see BDETERM).

mCAT V2.0 runs on NET/900+, H, and H+ (hence on all products using these, like BDETERM, BDEGRAPH, DC900, BITAD2), NET-A7 -A7R and -A7M, ETH-BIT, TSM-CPUxxH2 and -ARMCPU, ECB-BIT900H2, PER-DinX and all IPC-BIT900 versions.

Documentation can be inspected (as Adobe pdf file) using the following link:
mCAT.pdf

All versions of mCAT support the BITBUS fieldbus including full RAC functionality (GBS), however, they are not limited to be used with BITBUS but can work stand alone with a monitor on an RS232 line. Using mCAT allows immediate access to all i/o ports in the slave from the master with no programming on the slave side.

Hardware requirements:

An 80x86 computer with a serial RS232 line.

Software requirements:

mCAT in EPROM or Flash-EPROM on target. MS- DOS 3.21 or later, TLCS900-ANSI-C 4.x. The distribution CD contains the TOSHIBA evaluation version of the ANSI-C-Compiler - restricted to 3000 lines of source code - electronic handbooks only and no support or updates.