The mCAT Realtime Operating System

-

This document covers

- mCAT Version 2.00-2.20 for TLCS900 Microprocessors.
- mCAT Version 2.20 for ARM Microprocessors.

Document Version 91 - 1. August 2008 Volker Goller

Content

Tables

Table of Figures

I. Installing mCAT

1. A Note to Experienced mCAT Users

Experienced mCAT 2.xx users should read the mCAT 2.20 release notes first!

2. What You Need

1. A MCAT 2.20 Installation CD
2. A RS232 Cable
3. A piece of hardware that has mCAT 2.20 installed
4. A PC running Microsoft® Windows® 2000 or Windows® XP.

3. Install the Software

1. Insert the MCAT Installation CD into your CD-ROM drive
2. Start *setup.exe* found on the CD.
3. Follow the instructions. Your serial number can be found on a sheet that comes with the CD.
4. If the installation of mCAT is finished, you are asked to install additional products. The least you need is the GNUDE GCC compiler. You may install ADOBE® Acrobat Reader® to view the documentation online. We also recommend to install the UltraEdit® Programmers Editor. Be aware that UltraEdit is shareware and that the UltraEdit to come with mCAT is a 30 days trial version only. If you like UltraEdit register it at www.ultraedit.com.
5. If you press finish, the installation programs of the individual software packages you selected will be started. Please follow the instructions.

4. Where Can I Find mCAT

Setup.exe creates a shortcut folder named “mCAT2.20” (if you did not change the name) in your computers Start Menu folder. There you find a link to the this documentation in PDF format, a link to a index that will guide you to the example code, a shortcut to start an mCAT command shell and a shortcut to the wLGO Terminal Program you need to communicate with the mCAT hardware. See the figure below.

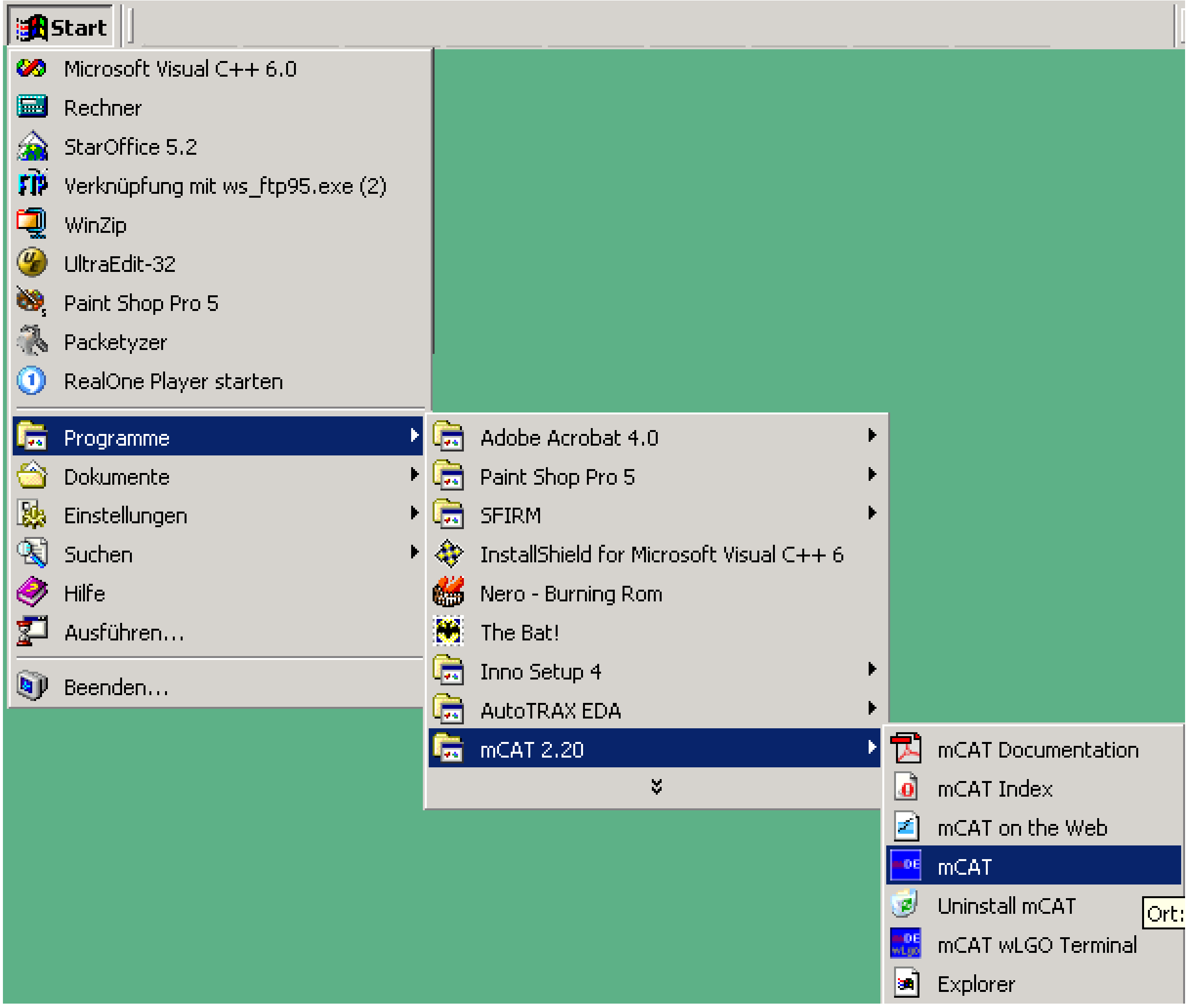


Figure 1: The mCAT Program Shortcuts

5. Get Connected

Connect the mCAT 2.20 Hardware (TSMARMCPU) with your PC using the RS232 cable. Prefered and pre-installed comport to use is **COM1**. On the mCAT hardware, use SER0. Start the wLgo Terminal-program from the “mCAT 2.20” Start Menu folder. If your mCAT hardware is powered up and you are connected to the correct serial port you will see mCAT's start up messages and finally you will get a SYSMON command prompt.

If you have to connect to another comport, you can change the port to use in the WLGO Menu “Einstellungen->Schnittstelle”.

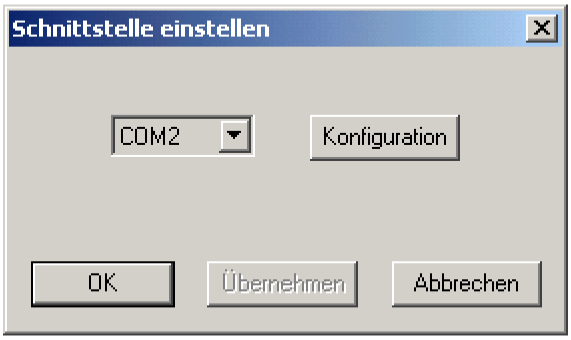
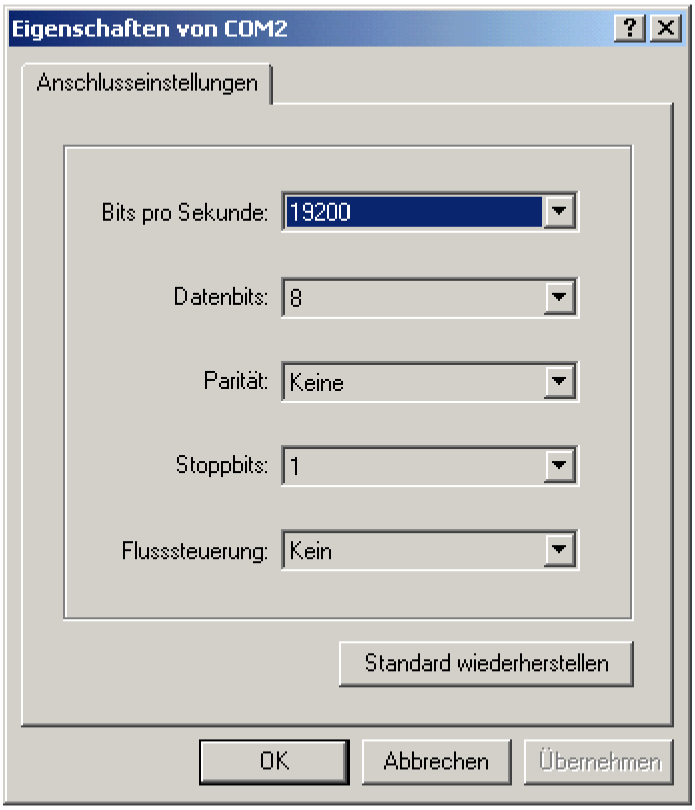


Figure 2: wLGO, Changing the COMPORT

If you have no success, check the WLGO setup. We need 19200-8N1, no handshake to communicate with mCAT. You will find the setup at “Einstellungen->Schnittstelle->Konfiguration” (the Button shown in the figure above).



Bits pro Sekunde: 19200

Datenbits: 8

Parität: Keine

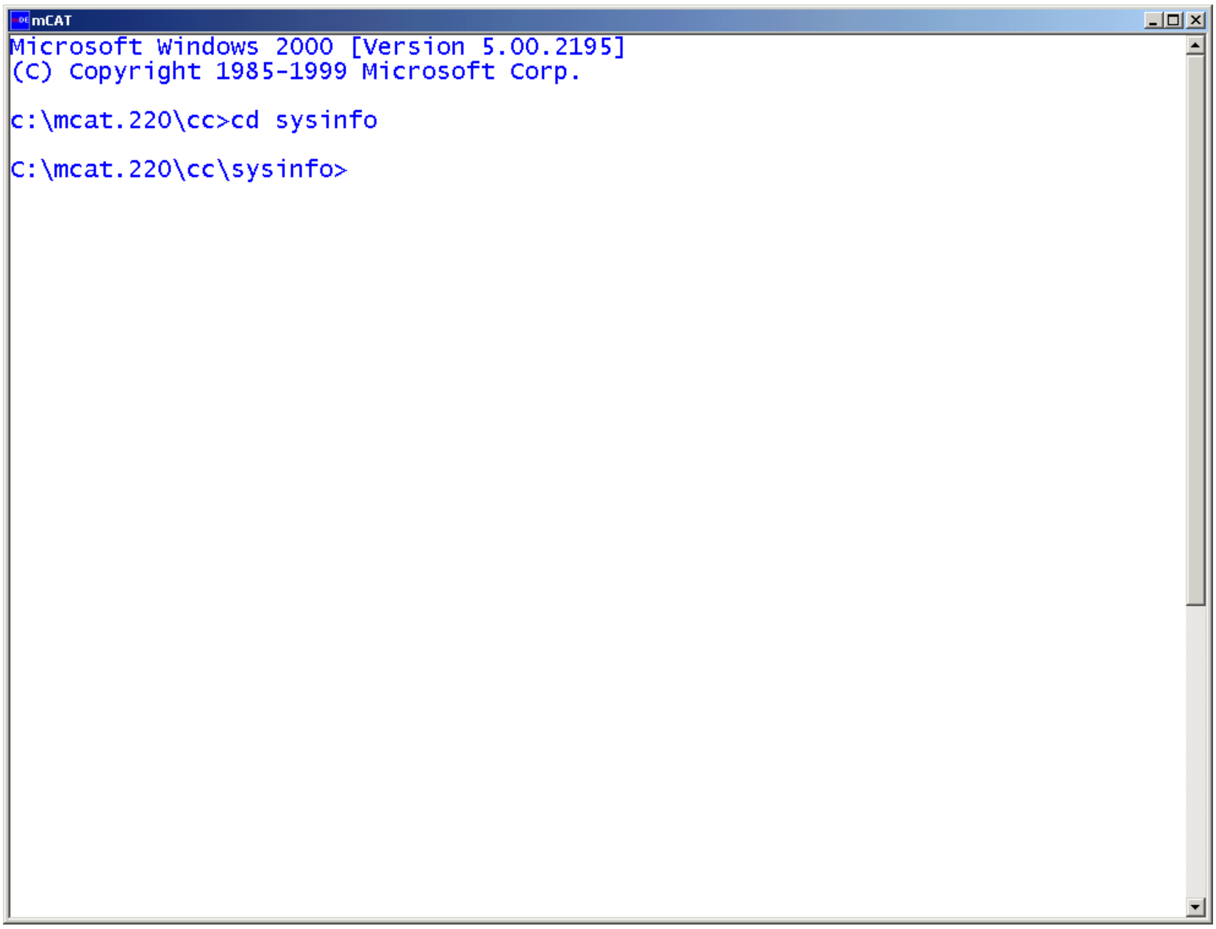
Stoppbits: 1

Flusssteuerung: Kein

6. Try an Example

Open the mCAT command shell and change to directory ***sysinfo***.

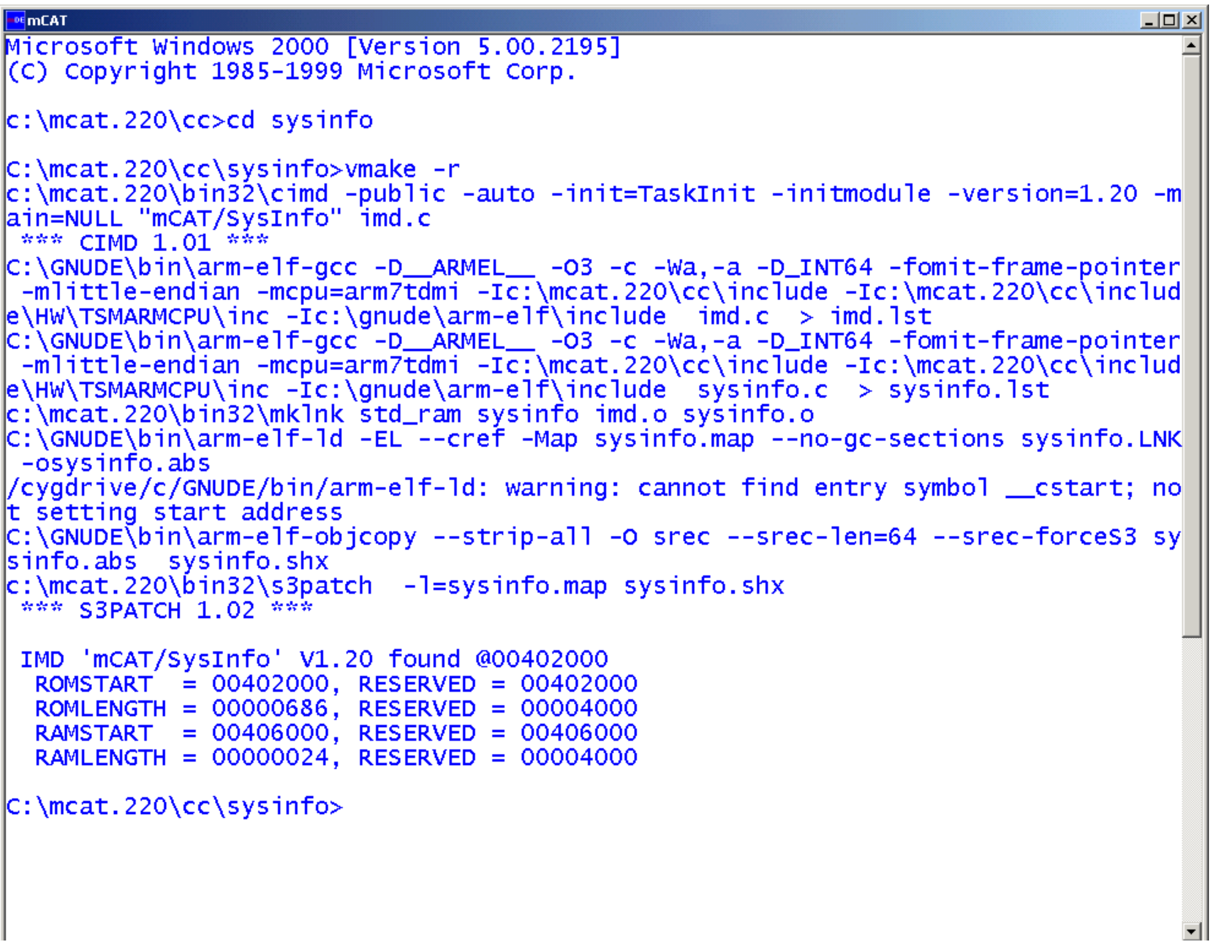
SysInfo is a limited example and for now it is not important what its function is.



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

c:\mcat.220\cc>cd sysinfo

C:\mcat.220\cc\sysinfo>
```

Figure 3: The mCAT command shell

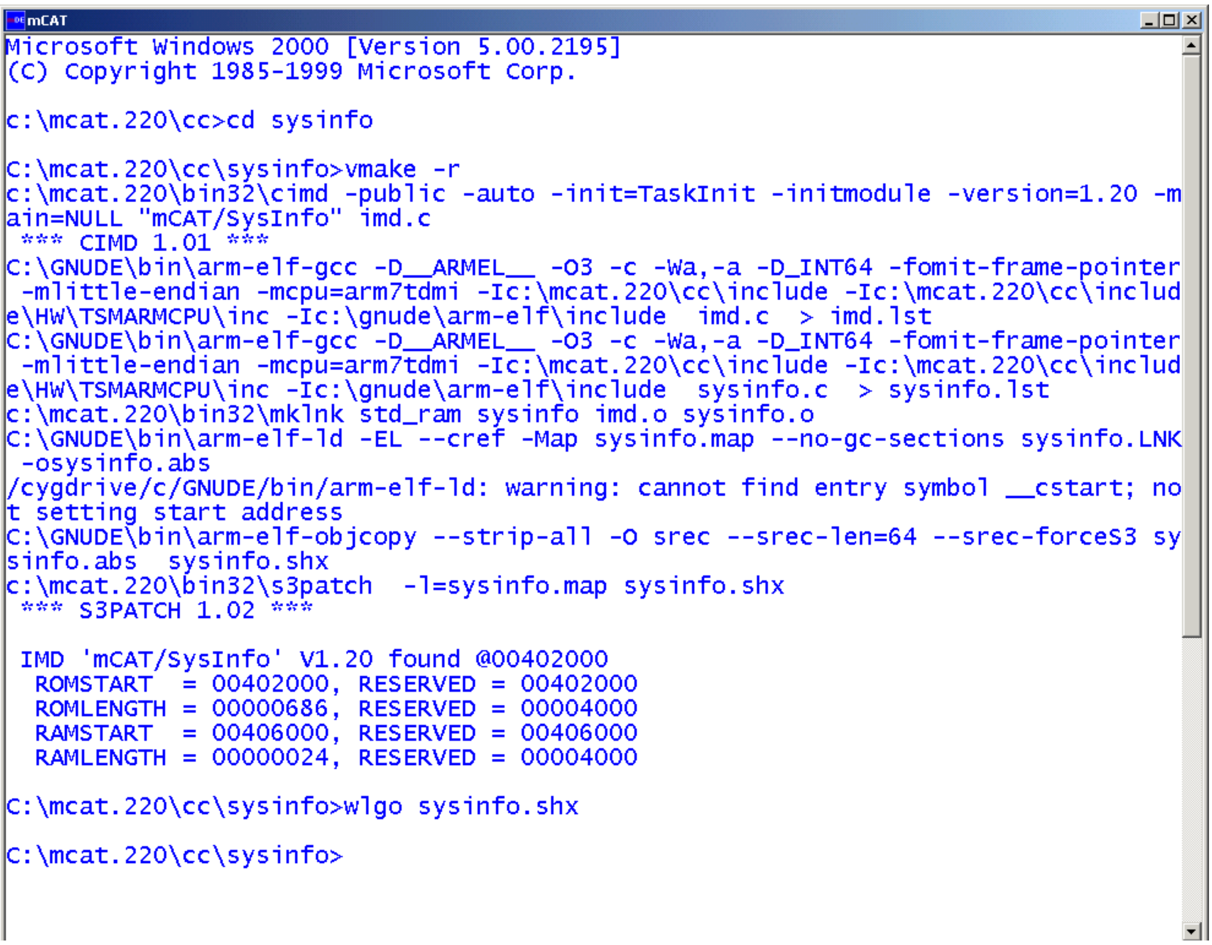
Enter ***vmake -r***. The system shall compile the sysinfo example. On success, the screen will look like this:



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

c:\mcat.220\cc>cd sysinfo

C:\mcat.220\cc\sysinfo>vmake -r
c:\mcat.220\bin32\cimd -public -auto -init=TaskInit -initmodule -version=1.20 -m
ain=NULL "mCAT/SysInfo" imd.c
 *** CIMD 1.01 ***
C:\GNUDE\bin\arm-elf-gcc -D__ARMEL__ -O3 -c -Wa,-a -D_INT64 -fomit-frame-pointer
 -mlittle-endian -mcpu=arm7tdmi -Ic:\mcat.220\cc\include -Ic:\mcat.220\cc\includ
e\HW\TSMARMCPU\inc -Ic:\gnude\arm-elf\include  imd.c  > imd.lst
C:\GNUDE\bin\arm-elf-gcc -D__ARMEL__ -O3 -c -Wa,-a -D_INT64 -fomit-frame-pointer
 -mlittle-endian -mcpu=arm7tdmi -Ic:\mcat.220\cc\include -Ic:\mcat.220\cc\includ
e\HW\TSMARMCPU\inc -Ic:\gnude\arm-elf\include  sysinfo.c  > sysinfo.lst
c:\mcat.220\bin32\mklnk std_ram sysinfo imd.o sysinfo.o
C:\GNUDE\bin\arm-elf-ld -EL --cref -Map sysinfo.map --no-gc-sections sysinfo.LNK
 -osysinfo.abs
/cygdrive/c/GNUDE/bin/arm-elf-ld: warning: cannot find entry symbol __cstart; no
t setting start address
C:\GNUDE\bin\arm-elf-objcopy --strip-all -O srec --srec-len=64 --srec-forceS3 sy
sinfo.abs  sysinfo.shx
c:\mcat.220\bin32\s3patch  -l=sysinfo.map sysinfo.shx
 *** S3PATCH 1.02 ***

 IMD 'mCAT/SysInfo' V1.20 found @00402000
  ROMSTART  = 00402000, RESERVED = 00402000
  ROMLENGTH = 00000686, RESERVED = 00004000
  RAMSTART  = 00406000, RESERVED = 00406000
  RAMLENGTH = 00000024, RESERVED = 00004000

C:\mcat.220\cc\sysinfo>
```

Figure 4: A Successful Compilation

Now, use wLGO to download the resulting SHX-File to the mCAT 2.20 hardware. There are two options with wLGO, you can use **F3** function key to open a file dialogue to locate and open sysinfo.shx or – sometimes more suitable – pass the shx file as a command line argument. Please note that wLGO does not accept a command line argument if it is already active. In this case, terminate wLGO.



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

c:\mcat.220\cc>cd sysinfo

C:\mcat.220\cc\sysinfo>vmake -r
c:\mcat.220\bin32\cimd -public -auto -init=TaskInit -initmodule -version=1.20 -m
ain=NULL "mCAT/SysInfo" imd.c
 *** CIMD 1.01 ***
C:\GNUDE\bin\arm-elf-gcc -D__ARMEL__ -O3 -c -Wa,-a -D_INT64 -fomit-frame-pointer
 -mlittle-endian -mcpu=arm7tdmi -Ic:\mcat.220\cc\include -Ic:\mcat.220\cc\includ
e\HW\TSMARMCPU\inc -Ic:\gnude\arm-elf\include  imd.c  > imd.lst
C:\GNUDE\bin\arm-elf-gcc -D__ARMEL__ -O3 -c -Wa,-a -D_INT64 -fomit-frame-pointer
 -mlittle-endian -mcpu=arm7tdmi -Ic:\mcat.220\cc\include -Ic:\mcat.220\cc\includ
e\HW\TSMARMCPU\inc -Ic:\gnude\arm-elf\include  sysinfo.c  > sysinfo.lst
c:\mcat.220\bin32\mklnk std_ram sysinfo imd.o sysinfo.o
C:\GNUDE\bin\arm-elf-ld -EL --cref -Map sysinfo.map --no-gc-sections sysinfo.LNK
 -osysinfo.abs
/cygdrive/c/GNUDE/bin/arm-elf-ld: warning: cannot find entry symbol __cstart; no
t setting start address
C:\GNUDE\bin\arm-elf-objcopy --strip-all -O srec --srec-len=64 --srec-forceS3 sy
sinfo.abs  sysinfo.shx
c:\mcat.220\bin32\s3patch  -l=sysinfo.map sysinfo.shx
 *** S3PATCH 1.02 ***

 IMD 'mCAT/SysInfo' V1.20 found @00402000
  ROMSTART   = 00402000, RESERVED = 00402000
  ROMLENGTH  = 00000686, RESERVED = 00004000
  RAMSTART   = 00406000, RESERVED = 00406000
  RAMLENGTH  = 00000024, RESERVED = 00004000

C:\mcat.220\cc\sysinfo>wlgo sysinfo.shx

C:\mcat.220\cc\sysinfo>
```

Figure 5: Loading a SHX-File from the ComamndLline

WLGO will start to download the shx and when this is done, the SYSMON command prompt is available again:

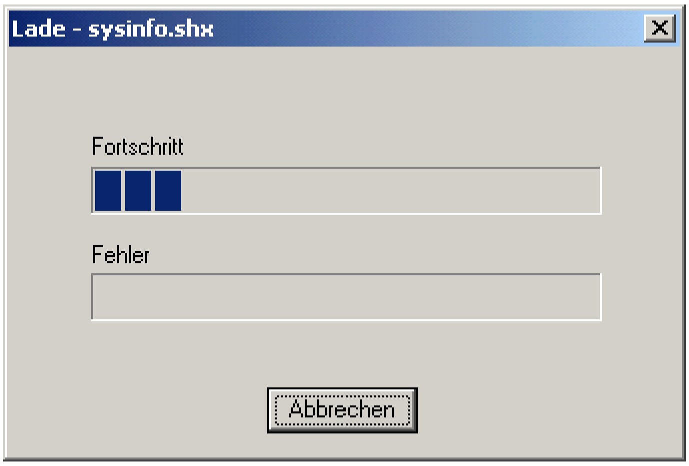
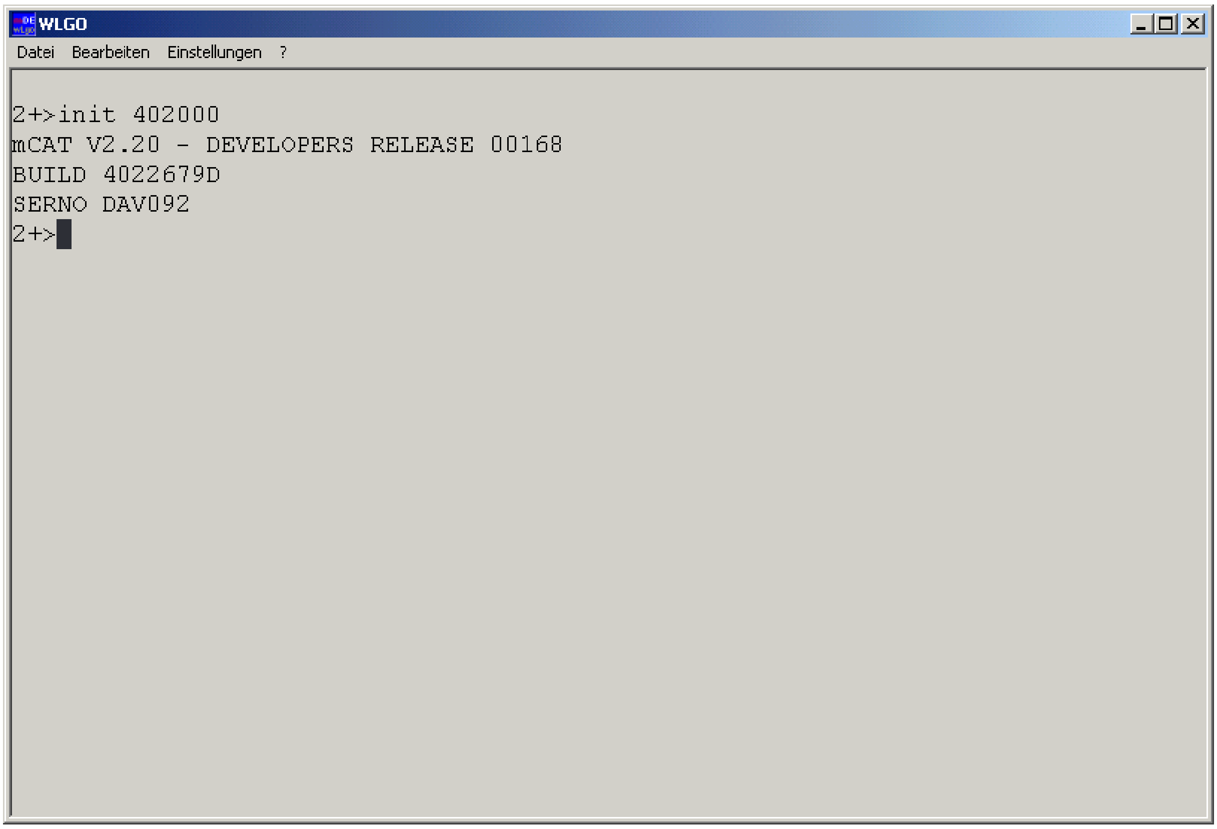


Figure 6: wLGO Download Progress Bar

SysInit is (like many other examples) compiled to be located in the user RAM (0x402000) Now, we can start our little example:

2+> init 402000



```
2+>init 402000
mCAT V2.20 - DEVELOPERS RELEASE 00168
BUILD 4022679D
SERNO DAV092
2+>
```

Figure 7: The Output from Our Example

The example prints the mCAT version and serial numbers as well as a build or the date of build. **Those values will be different at your system!**

7. DONE! The Things to do Next ...

If your installation works, you should start reading the mCAT UsersGuide, study the examples and consider the release notes.

II. mCAT 2 Users Manual

1. Introduction

MCAT is a modern real-time operating system designed to fill the gap between operating-system-less designs (*os-less*) and full size operating systems. It has a low to moderate memory footprint and allows fast task switching.

MCAT is a *multi-tasking* operating system, that relies on *message passing* to implement inter-task communication as well as synchronization. In this *Users Manual*, we will give an introduction to *multi-tasking* and *message passing* as well as to other major components and concepts of the mCAT system.

1.1. A few Tips

There are some simple rules that may help to get your fingers around mCAT and that may also help to successfully use mCAT.

- Don't try to get around the mCAT concepts. If you try to design your own components or applications using tricks and bits outside the mCAT concept, you *will* get unreliable and faulty results. That will also make it very difficult to *support* you.
- Do not take something for granted! If you use a new feature, read the manual before you start coding.
- If you have to implement complex calculation or algorithms, place them into a separate file and develop and debug them using a popular **C**-Compiler on your PC before you integrate it into mCAT. That can help to minimize in-system debugging and can speed up development time.

1.2. What You Need to Know Beyond mCAT

ANSI-**C** is the language used to develop mCAT components and applications. If you are new to **C**, you can learn it along with mCAT. However, our manuals do not give an introduction to **C**. There are many good books about **C** out on the market and you may even find some useful tutorials for free on the Internet (One example available at the time this manual was written: http://www.strath.ac.uk/IT/Docs/Ccourse/ccourse.html).

The mCAT manuals also do not give an introduction to basic software design knowledge, for example a “how to implement serial communication well and reliable”.

Finally, you need some time to read the mCAT manuals, investigate the examples and to get familiar with the mCAT concepts. *No investment, no return!*

2. The Elements of mCAT

And here they are, the elements of mCAT, in order of appearance:

- mCAT Modules – the basic assemblies
- mCAT Threads – the basic executable program units
- mCAT Tasks – the application process
- mCAT Message Passing – connects tasks with tasks and interrupt drivers with tasks.
- mCAT Interrupt Driver – control and handle hardware interrupts
- mCAT Shared Libraries – offer interfaces to shared function libraries
- mCAT Ticker Service – an universal timer service
- mCAT ExpressIO™ – an abstract process i/o layer

2.1. Modules

A *module* is any piece of binary data, usually – but not necessarily – executable program code. It can be compiled independently from other modules in a system. It is prefixed by a data structure called the ***IMD*** *(Initial Module Descriptor).* The most important element of this structure is a pointer to the executable function *TaskInit*.

The mCAT operating system scans the entire FLASH memory of a system at startup. For each valid module found, it invokes the modules TaskInit() function call. The name may confuse a bit, but TaskInit is not only the place to create and *init* a *task* but also to create and init an interrupt driver or any other kind of module. The name TaskInit is just used for historical reasons. *ModuleInit()* might have been a better choice. Anyway, TaskInit is a user provided function. We learn more about this function later.

> *Please note that TaskInit usually is not directly inserted into the IMD when a program is written in C. IMD holds a pointer to the c-compiler startup routine (runtime memory initialization, __cstart()) that will call TaskInit after it has completed its duties.*

The IMD holds a lot of information, including:

- Module name

- Module version
- Module build time (binary, UNIX time format)
- Module memory requirements (mCAT 2.10R00185 and higher)

The name and version is printed to the system console at startup.

> *Please note that mCAT itself consists of a set of independent modules.*

2.2. Threads

A *thread* is an independent program that can be executed semi-concurrently with other *threads*. With the current version of mCAT, the *thread* with the highest priority runs until it enters either a wait condition (via functions *MsgWait(), ThreadSleep()* or *ThreadSleepQueued()* or *ThreadDelay()*) or another higher priorized *thread* becomes ready to run (by leaving a wait condition). It is important to know the concept of *threads,* even if until you have a very advanced application you may find no need to deal with threads directly. You will find all details about threads in the mCAT Kernel Reference.

2.3. Tasks

Threads are restricted to exist within a *task* only. A Task is an mCAT module that is addressable by the *mCAT Message Passing* (see below) and that hosts at least one thread (the *main thread*). *Tasks* are the most important concept to implement an application using mCAT.

Tasks are globally addressable using their *task id* or *task number* (two names for the same thing). It is an integer number in the range of 0..MAX_TASKS (MAX_TASKS is 16 while this documentation is written). An mCAT task also has a name (up to 16-Char). A task is the frame for an application program.

Writing an mCAT application means to write a task in 98% of all cases.

That is all you need to know about tasks up to now. We will come back to tasks and application design in chapter I.3. Putting it all Together.

2.4. The mCAT Message Passing

Message Passing is a technique to implement interprocess communication (in mCAT: *inter-task communication*). A message can be assumed as a data structure holding user information that is passed from one task to another. To be useful, a message must have properties to:

- Identify the sender
- Identify the receiver
- identify the type of message (and thereby the type of user data included)

Optionally it should also carry the length of the message and an error indicator.

A simple real-world example may be helpful at this point:

Lets assume we have 2 persons ('*tasks*'): Alice and Bob. Bob works in a store and Alice needs something from Bobs store – lets say *real* Java beans.

1. Alice takes a long deep look into the yellow pages. She picks Bobs store.
2. Alice dials the number of Bobs store.
3. Bob answers the phone. He identifies himself. “Hi, here is Bob!”
4. Alice also identifies herself: “Hi, my name is Alcie. I like to order 1kg of your best Java beans! My address is BeachHouse, Halfmoon Bay”.
5. Bob Acknowledes the order “Well, 1kg of our finest Java! To Alice at Halfmoon Bay. Thank you for the order!”.
6. Now they can quit the phone conversation. The Java will be delivered the other day.

So what can we say about Alice and Bob and their conversation?

2.4.1. Bob

Lets start with Bob. We can say about Bob that he:

- Offers a service (Bobs store, selling Java).
- He has a phone number and this number is published in a common directory service (Yellow pages). We can say Bob is *addressable.*
- His major job is waiting for incoming calls (*orders*).

All these will qualify Bob to be a *server*. A server *offers a service*, has that service listed in a *publicly available directory service* and lingers most of his time waiting for incoming *orders.*

2.4.2. Alice

What can we say about Alice?

- Alice needs a specific service (like many others too!)
- Alice uses a *publicly available directory service* to find a provider who is able to deliver the service Alice needs.
- Alice calls Bob to send him her order. We can say: She sends him a *message*!
- Alice finally receives Bobs service.

If Bob is a server in our model, Alice is the client. She needs Bobs service. She may not know Bob or Bobs phone number before she needs his service. The contact is made via a *publicly available directory service.*

2.4.3. mCAT Implementation of Clients and Servers

If we transfer the Alice and Bob example to mCAT, we have to design a server task and (at least) one client task. With mCAT, a message from the client to the server (the order in or Alice/Bob example) is called a *request*. The answer Bob upon an order is called a *reply*.

Note: The structure and contents of a message is defined by the designer of the specific service within the server.

2.4.3.1. MCAT Message ID

Every mCAT message has a specific *message id.* This type code, a 32-Bit integer, is created by mCAT at runtime and registered in a database. The key to that database is a readable string assigned by the designer of the service. A server uses the function *MsgIdCreate()*to register a message id (and doing so a service offered by the server) and a client uses *MsgIdQuery()* to search the database for a specific message id (a service). The information stored in the *message id database* includes all addressing and routing information the client needs to send a *request* to the *server*.

2.4.3.2. MCAT Messages

An mCAT message is a data structure starting with a common entry called the *message header.* Its data type is MSG. This structure is 22-Bytes long and holds all information needed to transport the message from source to destination. Beside other data values it includes:

- The message id value (in the field *type*)
- The message source task (in the field *src*, needed for to be able to send back a request to its requester)
- An error indicator
- Priority fields

2.4.3.3. Using Priorities

In contrast to our Alice & Bob example, mCAT is capable of assigning priorities to its messages. With a request, you can not only assign the priority the *request* should have but also the priority the *reply* should use on its way back to the client.

An mCAT task/thread receiving a message will inherit the priority of an incoming message and execute the desired service at this priority.

Note:

> *For many cases, the MsgUpdate() function simplifies priority handling for clients by implicitly inserting the clients current priority into the priority fields within the request. However, MsgUpdate() is not always usable. Refer to the* ***mCAT Kernel Reference*** *for details on priorities and the MsgUpdate() function.*

2.4.3.4. Memory Management and Message Passing

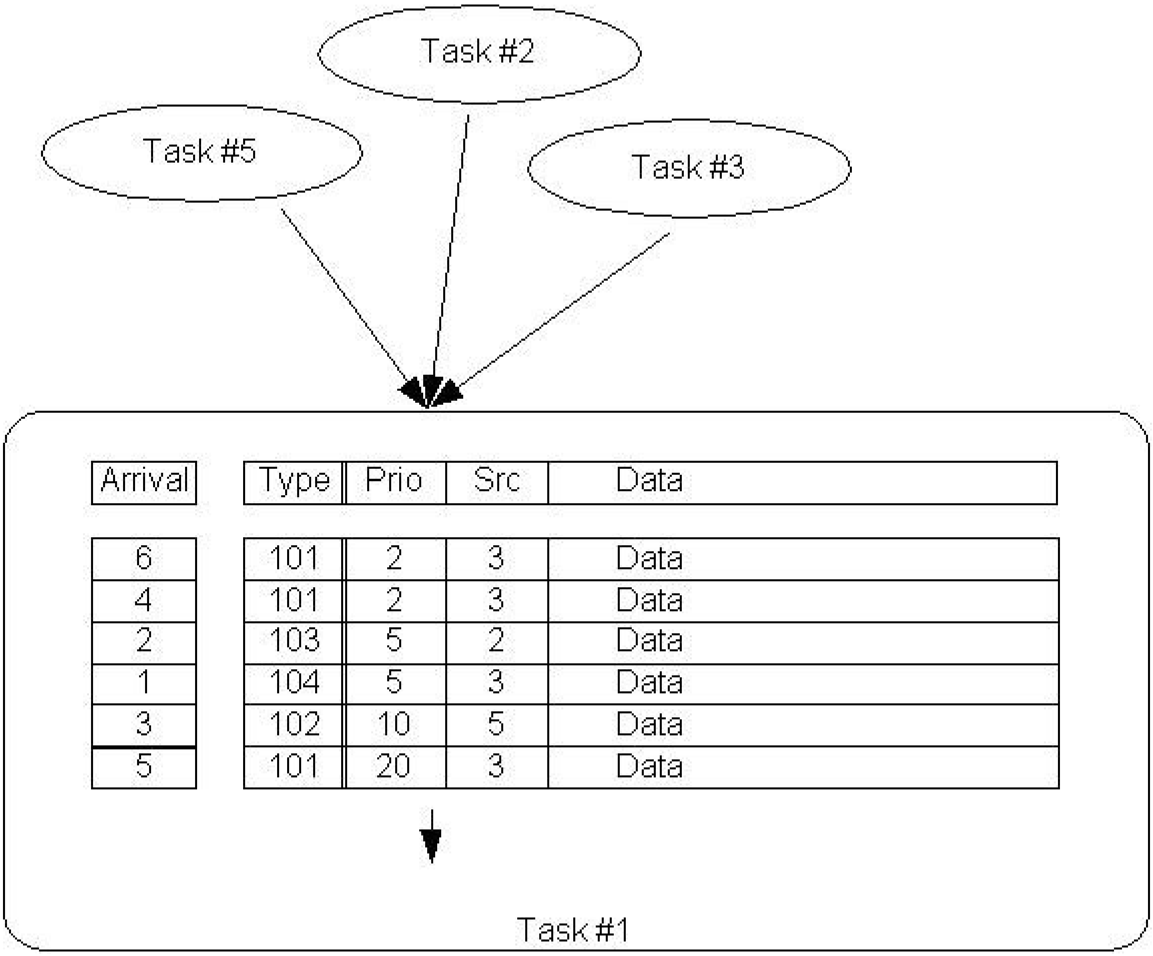
mCAT uses a *"store and forward message passing"* implementation. As a consequence, only a pointer to the message is actually passed from one task to another. No data is copied. The message is still a part of the senders dataspace.

A user *must* take this into account. If a statically allocated message is send to another task, the user *MUST* take care that this message is not modified by the sender until the sender receives the reply to this message. Again, the use of *MsgUpdate()* can make things simpler, because this function sends a *request* and waits until the related *reply* is received without consuming cpu time. This implies that there can be no memory usage conflicts using *Ms-*

gUpdate(). If *MsgUpdate()* is not used to implement the clients message interface it may make sense to use a *message pool.* A *message pool* is a construct to allocate fixed-length message buffers from a limited pool.

2.4.3.5. Message Queues and using Multiple Queues

An mCAT task or interrupt driver queues all incoming messages until they are read by the task. A task is usually designed around a central infinite loop, the message loop. First statement in that loop is a *MsgWait().* This function will try to retrieve a message from the message queue.

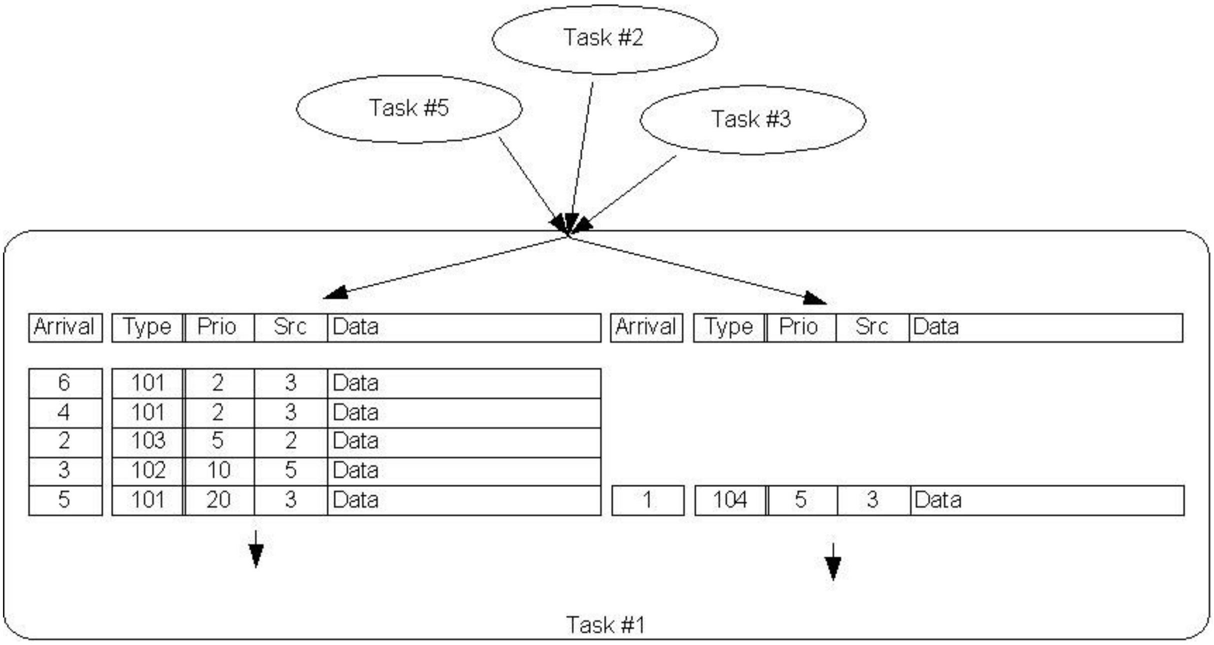


Arrival
6
4
2
1
3
5

Type	Prio	Src	Data
101	2	3	Data
101	2	3	Data
103	5	2	Data
104	5	3	Data
102	10	5	Data
101	20	3	Data

Figure 8: mCAT message queue

If at least one message is waiting in the queue, it is fetched and the current thread is switched to the messages priority (exception: If the task runs in fixed priority mode). If no message is pending, the task will terminate execution and sleep until either an optional timeout occurs or a message is queued. Please note that the messages are queued sorted by time **and** priority! The message with the highest priority is retrieved first. Messages with the same priority are sorted by time of arrival, oldest first.



Arrival	Type	Prio	Src	Data
6	101	2	3	Data
4	101	2	3	Data
2	103	5	2	Data
3	102	10	5	Data
5	101	20	3	Data

Arrival	Type	Prio	Src	Data
1	104	5	3	Data

Figure 9: A separate queue for message type 104

When it comes to practical application design, there can be a situation, when specific types of messages need to be handled in specific states of the application. Receiving them in the main loop, the programmer will have the need to store those messages *somewhere* until he needs them. An easier and more elegant way to handle such situations is to create an individual queue for those message types. All messages whose message types are not assigned to a specific queue are stored in the default queue.

2.5. Interrupt Drivers

MCAT supports a straight Interrupt Driver model. An interrupt module can handle interrupts natively and send mCAT messages in response to those events. They can also receive messages form both, tasks and other interrupt drivers. Writing interrupt drivers is a very advance project that is in good hands of an experienced mCAT user. The details about writing interrupt drivers can be found in the mCAT Kernel Reference.

2.6. Shared Libraries

Shared Libraries are also designed to offer different clients a service. In contrast to a server implemented in a concurrent task and interfaced by message passing, a shared library offers synchronous services only. That implies that they – usually - can not manage shared resources and handle multi-tasking (as usual there are exceptions, but only a few and they are hidden very deep in the system, like the Memory Management API).

However, a shared library can save resources. If there are functions and constant data used by several tasks, it is much more efficient to move this code into a shared library than to link the code with every task separately. So it will consume the memory it needs only once. If it needs to handle resource allocation, it may interact with a message based server by using the *MsgUpdate()* function.

Please note that the mCAT-Kernel itself is a shared library.

2.7. The Ticker Service

In most applications its necessary to do things in a fixed cycle or related to some timing constrains. Maintaining timeout control, regularly checking of a device state or sampling data – you need a timebase to implement those applications.

If an mCAT task needs such a service it can rely on the *Ticker*. Using a simple function *TALL()* the task requests the Ticker to periodically send a message to the task at a given frequency. Because the ticker message is a standard mCAT message it is easy to handle. Its just another *if-statement* in the central message loop of an application. Our first and widely referenced example *“TickTest”*. A Ticker-Example is for a realtime operating system what “*hello world”* is for desktop programming systems.

2.8. ExpressIO™

ExpressIO™ is an process I/O abstraction layer. It provides fast and lean methods to access process I/O like event counters, position encoders and analog or digital I/O. ExpressIO™ applications are portable if they are moved to different hardware – as long as the new hardware also supports ExpressIO™. In chapter *“The Gifts of ExpressIO”* we will present a simple ExpressIO™ example. More details about ExpressIO™ can be found in the documentation.

2.9. Memory Management

With MCAT we use a very simple *allocate-and-never-free* approach to memory management. This is driven by the believe that an embedded system should be designed in a way that dynamic memory allocation should be restricted to system startup configuration if possible.

For highly dynamic memory management requirements with limited flexibility, like message buffer management, we offer The *Message Pool* API. A message pool is a fixed size memory area that is splited into a fixed number of buffers of a unique length. Allocation and Deallocation of a buffer from a *Message Pool* is very fast.

2.10. C Programming Style and supplied MACROS

Working with our examples you will find that we predefined a few macros that are not mCAT specific and not common for C. Here is the place to introduce them:

2.10.1. PRIVATE, PUBLIC and EXTERN

If not labeled with the *static* attribute, all function names and global variables of a c-source file are exported and are visible to other files when the linker is finally putting it all together. That is not a very helpful feature of C and we believe it is a good idea to explicitly label ALL functions and global variables using our macros PUBLIC (=> export name) and PRIVATE (=> do NOT export name). I do not know who did this first but I picked the use of PRIVATE and PUBLIC from operating system designer Andrew Tanenbaum who used them in his MINIX operating system.

We also add the macro EXTERN It fixed some compiler limitations of an early Toshiba C-compiler. It is used in conjunction with PUBLIC. One file declares a variable or function public, another declares the same as EXTERN.

> ***Please note that PUBLIC and PRIVATE should NEVER be used to label local variables (variables on stack). Local variables are always private to the scope of the current function.***

2.10.2. Loop

With mCAT, most tasks will be designed around a central infinite loop that handles incoming messages. The C programming language does not support an infinite loop by a statement. Usually programmers use something like

```
        while (1) {
            // infinite loop
        }
```

to implement infinite loops. This reads not very intuitive. So we wrap this statement using a macro:

```
#define loop while(TRUE)
```

Loop can be used as for() or while() loops:

```
        loop {
                msg = MsgWait(0,0);
                if (msg->type == ticker->id) {
                        ....
        } /* endloop */
```

2.10.3. ARRAYSIZE

The size of an static array is hard to determine. The *sizeof* operator will return the size in bytes. This will not help if you need the size in terms of *items*. A simple example will show up the problem:

```
int my_array[7];
int my_function_sum_of_array()
{
        int     sum,i;
        for (sum=0,i=0;i<7;i++) {
                sum += my_array[i];
        }
        return sum;
}
```

The function *my_function_sum_of_array()* will add up all items in *my_array[]*. So far it works. However, if I change the array size, I have to change the function as well. One common solution to this problem is the use of static macros:

```
#define MY_ARRAY_SIZE      7
int my_array[MY_ARRAY_SIZE];
int my_function_sum_of_array()
{
        int     sum,i;
        for (sum=0,i=0;i<MY_ARRAY_SIZE;i++) {
                sum += my_array[i];
        }
        return sum;
}
```

This is much better. Changing the macro **MY_ARRAY_SIZE** will be sufficient. But there is an even smarter solution available:

```
int my_array[7];
int my_function_sum_of_array()
```

```
{
        int     sum,i;
        for (sum=0,i=0;i<ARRAYSIZE(my_array);i++) {
                sum += my_array[i];
        }
        return sum;
}
```

The macro *ARRAYSIZE()* will calculate the size of an array in items in a reliable and portable way. And the calculation will be done at compile time, so there is no penalty to pay for using it. And here is the implementation of ARRAYSIZE:

```
#define ARRAYSIZE(x)     (sizeof(x)/sizeof(x[0]))
```

2.10.4. ALIGN

ALIGN(val,al) returns a value that is multiple of *al* and not smaller than input value *val*. The macro works with any value of *al*, even if alignments to binary boundaries (2,4,8,..) are the most common usage of alignment operations. Modern compilers will detect binary boundary operation and optimize the code as needed.

```
#define ALIGN(val,al)    ((((val) + ((al)-1)) / (al)) * (al))
```

3. Putting it all Together

3.1. A Simple Example: TICKTEST

To get a first idea about how the actual code looks like, here's an example of a simple program that prints out a counter every 1/10 seconds to the terminal serial line. To accomplish this, it will request the ***ticker service*** to send a message every 1/10 second to the task.

3.1.1. Includes Needed

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 function definitions for TALL and TAFTER, while SIMPLEIO.H defines basic serial line i/o-functions. Since mCAT 210-R160 and later SIMPELIO provides std. C output functions *printf()* and *fprintf()* beside some basic functions like *WrStr()* used traditionally with mCAT. The only limitation to printf/fprintf implementation in SIMPELIO is that float point types (*double* and *float*) are not supported. To output those, use sprintf to format a buffer and send the buffer to the serial line using printf.

3.1.2. TaskInit

The INIT part of the task - *TaskInit()* - follows. Because there is no need for specific initialization, we just create and start the main task using TaskStartup().

TaskStartup creates a task, assigns 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 macros *Protect()* and *UnProtect()* are wrappers to *ThreadProtect()/ThreadUnProtect()*. The code between them can not be interrupted by a task switch. They are used to implement critical sections. It is recommended to handle *TaskInit()* as a critical section, because if the code is executed on behalf of SYSMON's ***init*** command for test purposes the behavior is unpredictable if it is not protected. If the task is moved to FLASH and *TaskInit()* is executed by the mCAT bootloader this protection is not needed – but it will not harm either. So its better to always handle this as a critical section.

3.1.3. TaskMain()

TaskMain() is the main part of the task, it would be main() in a standard C environment. First local variables are declared. Our variable *pock* is going to be the counter variable that we will print out as the number of messages received.

The TALL-function is parameterized to use the message tick to be sent all 100 milliseconds at a priority of 200 to my task.

The program continues as an endless loop. At the beginning 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 at 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 SYS_WAIT_INIFINITE 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 a NULL pointer in case of a timeout.

When MsgWait returns a message, the program has to check its type. As we are waiting for a message from ticker, the type field in the msg data structure (msg->type) must contain the ID of a *TickerMsg*. 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 printed 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.

3.1.4. The Source Code

```
/*
 *    ticktest
 *
 *    (c) 1999-2004 mocom software GmbH & Co KG
 *
 *    File: ticktest.c
 *
 *    History:
 *
 *     date        version author comment
 *    ----------------------------------------------------------------------
 *     20.09.1995  V1.00   VG     created
 *     13.04.1999  V1.01   VG     updated, new code to find sysmon
 *     02.06.2004  V1.10   VG     using SysmonEnable to deactivate/reactivate
 *                                SYSMON. Updated comments.
 *    ----------------------------------------------------------------------
 */
#define __MOD_TICKTEST

/*----------------------------------------------------------------------------*/
/* THIS FILE CAN BE COMPILED WITH TOSHIBA C, Metaware HighC and GCC for ARM   */
/*----------------------------------------------------------------------------*/

#include <mcat.h>       /* mCAT functions and datatypes */
#include <ticker.h>     /* ticker msg and functions */
#include <simpleio.h>   /* Simple, non standard io functions */
#include <ansi.h>       /* ANSI-Terminal control */
#include <string.h>     /* std. c striung functions */

PUBLIC  INTEGER     Self;

/*----------------------------------------------------------------------------*/
/* INIT TASK                                                                  */
/*----------------------------------------------------------------------------*/

void TaskInit (IMD *imd)
{
    INT16 error;            /* you may or may not check for errors.
                               There must be a fundamental problem if
                               the task will not start */
    Protect();              /* This is necessary to avoid problems while
                               init the task from within the monitor */
    Self = TaskStartup(imd,FromTop,&error,0);
                            /* imd      = pointer to own imd
                               FromTop  = allocate task number 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                                                */
/* The macro PRIVATE makes the symbols local to this file.                    */
/*----------------------------------------------------------------------------*/

PRIVATE TickerMsg tick;          /* used to request service by the ticker */

void InitScreen()
{
    // Disable sysmon using 'SysmonEnable(FALSE)'
    if (!SysmonEnable(FALSE)) {
        TraceWriteLog("CAN'T DISABLE SYSMON - EXIT\n");
        TaskDelete(Self);
    } /* endif */

    // clear screen and print banner
    clrscr();               /* CLEAR SCREEN */
    gotoxy(2,3);            /* SET CURSOR */

    printf(" *** TICKTEST 1.0 ***\n\n");
                                    /* hello */
    gotoxy(2,18);           /* STATIC TEXT */
    printf("Press any key to exit ...");
    gotoxy(13,10);
    printf("1/10 sec since start.");
}

void TaskMain ()
{
    lword pock;             /* a "tick counter" */
    MSGID *TickerId;        /* ID of tickermsg */
    MSG *msg;               /* a pointer to handle incoming messages */

    pock = 0l;              /* clear tick counter */

    TickerId = TALL(&tick,100l,200,Self);
                                    /* request the ticker to send "tick"
                                       all 1000ms (1sec) to ourself (Self)
                                       using a priority of 200 */
    if (TickerId == NULL) {
        TraceWriteLog("TALL failed\n");
        TaskDelete(Self);
    } /* endif */

    // setup screen
    InitScreen();

    /*--------------------------------------------------------------------*/
    /* Ensure that you NEVER return from this TaskMain function!          */
    /* The only way to exit is to call "TaskDelete(Self);"                */
    /*--------------------------------------------------------------------*/

    /* loop is a c-macro "#define loop while(1)". It is defined in
       "vtype.h", included by "mcat.h" */

    loop {
        /* using SYS_WAIT_INFINITE is portable between ARM & TLCS900.
           On TLCS900 platform 0 is used to signal no wait, on the
           ARM platform we use -1. */
        msg = MsgWait(0,SYS_WAIT_INFINITE);      /* wait for a msg. Timeout is NULL,
                                           use default queue "0" */

        /* we got a msg. Check if it was a ticker msg */
        if (msg && msg->type == TickerId->id) {
            /*--------------------*/
            /* IT'S A TICKER MSG  */
            /*--------------------*/
            MsgSendReply(&tick,Self,ACK);   /* Acknowledge FIRST! */
            gotoxy(8,10);
            printf("%d",++pock);
            if (kbhit() || pock > 150) {
                RdChar();
                clrscr();
                // reenable SYSMON and kill ourself.
                if (!SysmonEnable(TRUE)) {
                    SysReset();
                } /* endif */
                TaskDelete(Self);
            } /* endif */
        } else if (msg) {
            /*--------------------*/
            /* UNKNOWN MSG!       */
            /*--------------------*/
            MsgSendReply(msg,Self,NAK);
            puts(" *** FAIL ***\n");
        } /* endif */
    } /* endloop */
}
```

3.1.5. Creating the Makefile

A makefile is used to control the compilation process.

```
#
# Project: TICKTEST
# Author : VG
# Date   : 02.06.2004
#
PROJECT = TICKTEST

# memory spec
TARGET = std_ram

# optional c-compiler options
CMD=-D$(HARD)

# list of object files. File holding the IMD must be the first in the list.
OBJFILES = imd.$(REL) $(PROJECT).$(REL)

# list of include files
INCFILES =

# build rule, same with all projects
$(PROJECT).shx: $(OBJFILES)
	$(MKLNK) $(TARGET) $(PROJECT) $(OBJFILES)
	$(LD) $(PROJECT).LNK -o$(PROJECT).abs
	$(CONVERT) $(PROJECT).abs $(OF) $(PROJECT).shx
	$(S3PATCH) -l=$(PROJECT).map $(PROJECT).shx

# individual dependencies
$(PROJECT).$(REL):      $(PROJECT).c    $(INCFILES)

# build rule for 'imd.c', rebuild when makefile was changed.
# cmd {-options} <name> <file.c>
imd.c:              makefile
	$(CIMD) -public -auto -init=__cstart -version=1.10 mCAT/TickerTest imd.c
```

3.1.5.1. Selecting a Target Memory Description

One important question to answer is: Where to place my code & data? MCAT does not manage this on its own.

For non-complex projects, this is easy to answer. MCAT reserves an area in RAM and FLASH for small projects. Using RAM, starting at 0x402000, it is easy to download and test modules. No FLASH specific handling (deleting) must be considered. Moving the project to FLASH gives it the ability to auto-start after reset.

In more complex projects, each module may reside in a different FLASH page. Each module needs separate areas in the RAM space to store variables.

MCAT use so-called TARGET files to control the address settings. A sample TARGET file looks like:

```
[ADDR]

ROMSTART=0x800000
RAMSTART=0x412000

ROMLENGTH=0x10000
RAMLENGTH=0x10000
```

Please note that ROMSTART/ROMLENGTH are used to define an area to store constant data (program code and constant values). RAMSTART and RAMLENGTH are used to define an area to store dynamic data (variables). The names do not determine that the ROMXXXX section MUST be located in FLASH memory! For testing, this can be located in RAM as well:

```
[ADDR]

ROMSTART=0x402000
RAMSTART=0x412000

ROMLENGTH=0x10000
RAMLENGTH=0x10000
```

The memory definition is set in the makefile using the macro TARGET. The value set is the TARGET files name without extension (extension must be ***.trg***). The above examples are the standard TARGET files ***std_ram*** & ***std_rom***, used to cover simple projects and provided along with mCAT. All examples provided use one or both of these TARGET files:

```
# memory spec
TARGET = std_ram
```

3.1.5.2. Generating the Initial Module Header (IMD)

The Initial Module Header is generated by a tool called ***CIMD.EXE***[1]. This tool sets up all fields of the structure and calculates a proper hash code. The hash code is calculated from the modules name. Together with the pattern AA 55 (a 16-bit integer, 0x55aa, little endian) the hash code is used to verify the validity of an IMD.

The IMD is written into a separated file. This file should not be edited manually! It looks like this:

```
// Generated by CIMD.EXE
// Do not edit manually

#include <mcat.h>
```

[1] With previous versions of mCAT a tool called IMD.EXE was used to generate IMD's. This tool is obsolete, do not use it anymore

```

INTEGER TaskMain (void);
INTEGER __cstart (IMD *imd);

#define VERSION          "1.10"
#define NAME             "mCAT/TickerTest"

#define PATTERN          0x55aa
#define ID               0xff
#define PRIORITY         128
#define MAINCALL         TaskMain
#define INITCALL         __cstart
#define IMDNAME          taskimd
#define MODE             MODE_TASK
#define HEAP             0
#define CHECKSUM         29391
#define SCOPE            PUBLIC
#define STACK            1024  // 0x400
#define BUILD            0x112d2d11

SCOPE const IMD IMDNAME = {
  PATTERN,INITCALL,MAINCALL,STACK,
  HEAP,BUILD,PRIORITY,MODE,
  ID,VERSION,NAME,CHECKSUM
};
```

The advantage of generating a C-Source file is that this file is compatible across the mCAT platforms (ARM,TLCS900). Please note that almost any option in the header can be changed by means of CIMD.EXE arguments:

```
 *** CIMD 1.01 ***
cimd {options} <name> <imdfile.c>

options are:
 -public         : imd structure is public (default private)
 -auto           : autostart option, default is 'no autostart'
 -interrupt      : setup imd mode MODE_INTERRUPT (default=MODE_TASK)
 -initmodule     : setup imd mode MODE_INIT (default=MODE_TASK)
 -version=#.##   : set initial version, default=1.00
 -priority=###   : set initial priority to ### (0 < priority <255), default=128
 -stack=####     : set stack size, default is 1024
 -init=<name>    : name of the init call, default is __cstart
 -main=<name>    : name of the main call, default is TaskMain
 -imd=<name>     : imd structure name, default is taskimd
```

For example, if the option ***-auto*** is not given, the pattern AA 55 is not written to the structure. Instead the pattern FF 55 is used. The module is not auto start able in that case. However, using SYSMON's ***val*** command it can be changed into an auto-start module at runtime if it is located in FLASH memory. See SYSMON documentation for more information on ***val***.

CIMD should be integrated into a projects ***makefile***. The generated object file must be the first in the link sequence. In our example, we use the filename imd for the generated file (imd.c, imd.$(REL)):

```
# list of object files. File holding the IMD must be the first in the list.
OBJFILES = imd.$(REL) $(PROJECT).$(REL)
```

It is recommended to make the imd source file (***imd.c*** in our case) depending of the makefile itself. So we force a re-generation whenever an argument to CIMD is changed:

```
# build rule for 'imd.c', rebuild when makefile was changed.
# cmd {-options} <name> <file.c>
imd.c:             makefile
      $(CIMD) -public -auto -init=__cstart -version=1.10 mCAT/TickerTest imd.c
```

3.1.6. Compile and Download

To compile TICKTEST, follow the steps:

- Open mCAT command line
- Change to the directory where **TICKTEST** is located (***<mcat_dir>\cc\ticker***)
- enter vmake

The resulting output should look like:

```
C:\mcat.220\cc\ticker>vmake
C:\GNUDE\bin\arm-elf-gcc -D__ARMEL__ -O3 -c -Wa,-a -D_INT64 -fomit-fra
me-pointer -mlittle-endian -mcpu=arm7tdmi -Ic:\mcat\mcat\arm\cc\include -Ic:\mca
t\mcat\arm\HW\TSMARMCPU\inc -Ic:\gnude\arm-elf\include -DTSMARMCPU TIC
KTEST.c  > TICKTEST.lst
c:\mcat\bin32\addfile TICKTEST.cll TICKTEST.c
C:\GNUDE\bin\arm-elf-gcc -D__ARMEL__ -O3 -c -Wa,-a -D_INT64 -fomit-fra
me-pointer -mlittle-endian -mcpu=arm7tdmi -Ic:\mcat\mcat\arm\cc\include -Ic:\mca
t\mcat\arm\HW\TSMARMCPU\inc -Ic:\gnude\arm-elf\include -DTSMARMCPU imd
.c  > imd.lst
c:\mcat\bin32\addfile TICKTEST.cll imd.c
c:\mcat\bin32\mklnk std_ram TICKTEST TICKTEST.o imd.o
C:\GNUDE\bin\arm-elf-ld -EL --cref -Map TICKTEST.map --no-gc-sections
TICKTEST.LNK -oTICKTEST.abs
C:\GNUDE\bin\arm-elf-objcopy --strip-all -O srec --srec-len=64 --srec-
forceS3 TICKTEST.abs  TICKTEST.shx
c:\mcat\bin32\s3patch -s=240 -l=TICKTEST.map TICKTEST.shx
 *** S3PATCH 1.02 ***

 IMD 'mCAT/TickerTest' V1.10 found @00402000
  ROMSTART  = 00402000, RESERVED = 00402000
  ROMLENGTH = 0000090A, RESERVED = 00010000
  RAMSTART  = 00412000, RESERVED = 00412000
  RAMLENGTH = 00000094, RESERVED = 00010000
```

```
C:\mcat.220\cc\ticker>
```

To download the resulting image file (***ticktest.shx***) into the mCAT node, use ***wlgo.exe:***

```
C:\mcat.220\cc\ticker>wlgo ticktest.shx

C:\mcat.220\cc\ticker>
```

Note: If *wlgo* is already running, it will NOT start the download process. In this case, exit *wlgo* or open the download dialog using the key **F3**.

3.1.7. Execute and Debug

Once the program is successfully loaded, it can be started using the **SYSMON** command

init 402000.

This command executes a modules *TaskInit()* function of the module located at 402000 – and that is the usual location for RAM based test programs and the location we choose for TICKTEST.

TICKTEST will clear the Terminals screen and display the counter:

To exit TICKTEST, press any key.

Great! To terminate the session, the reset command is your choice:

```
2+>reset
```

3.2. A Example using ExpressIO

ExpressIO is an abstract interface to the process i/o hardware. It is an important part of the mCAT programming environment.

3.2.1. The Gifts of ExpressIO

ExpressIO makes it easy to structure and access physical process i/o. It allows the creation of i/o objects that may inherit software implemented features (called ***ExpressPrograms***) not native to the physical i/o. Those features include counters, edge detectors and pulse generators.

With ExpressIO one can address a physical i/o directly using its physical location information:

- The **channel** number of a
- **module** that is attached to a specific
- **bus**.

This triplet ***bus.module.channel*** can also be used to create an ExpressIO IOOBEJCT. Once an IOOBEJCT is created, one can access i/o using the object and no longer worrying about the triplet. Accessing all i/o in a program via IOOBEJCTS makes it easy to port, adapt and enhance a program. To port it to a different hardware or to route a i/o to a different physical port, just the ***IOObjCreate()*** function has to be changed. To make this even more convenient, it is recommended to keep the creation and configuration of IOOBJECTS in a single function at the beginning of your program. The recommended name of this functions is SYSTEM().

In our example, you will see how to create an IOOBJECT, how to inherit a ExpressProgram, how to subscribe events generated by the ExpressProgram. Running the program from SYSMON will show how useful it is to have both options: addressing a raw i/o via the triplet bus.module.channel and via the named IOOBJECT.

3.2.2. The Source Code

```
/*
 *    EDGE
 *
 *    (c) 2004 mocom software GmbH & Co KG
 *
 *    File: EDGE.C
 *
 *    Created: 02.06.2004  11:11:06  VG
 *    ----------------------------------------------------------------------
 *
 *    History:
 *
 *     date     version     author      comment
 *    ----------------------------------------------------------------------
 *     02.06.2004  V2.00       VG       derivated from an older version
 *    ----------------------------------------------------------------------
 */
#include <mcat.h>
#include <xio\express.h>
#include <simpleio.h>
#include <ansi.h>
```

```
// needed for C startup code
PUBLIC  INTEGER     Self;

/*--------------------------------------------------------------------------
 *    file-local variables
 *--------------------------------------------------------------------------
 */
PRIVATE IOOBJECT em_stop;   /* input, emergency stop */


/*--------------------------------------------------------------------------
 *    ExpressIO initialization function.
 *--------------------------------------------------------------------------
 */
INTEGER SYSTEM ()
{
    INTEGER error;

    // create IOObject and store error code. We bind the i/o port
    // BUS_TYPE_CPU,CPU_DIN,0 (aka CPU.1.0) to a named IOOBEJCT.
    // The object inherits the EDGE detector ExpressProgram.
    error = IOObjCreate (&em_stop,              // the ExpressIo object
                         "EMERGENCYSTOP",       // a human-readable name
                         BUS_TYPE_CPU,          // select a bus (CPU|TSM|I2C...)
                         CPU_DIN,               // select a module (CPU_DIN)
                         0,                     // select a channel (I1 on CPU)
                         CLASS_DIGITAL,         // we need a digital input!
                         "EDGE");               // inherit a edge detector

    // display error code on fail
    if (error != IOERR_OK) {
        printf("IOERR = %d\n",error);
        return FALSE;
    } /* endif */

    // great, we created an IOOBEJCT!
    return TRUE;
}

/*--------------------------------------------------------------------------
 *    Task initialization function.
 *--------------------------------------------------------------------------
 */
void TaskInit (IMD *imd)
{
    short error;

    Protect();
    Self = TaskStartup(imd,FromTop,&error,0);
    UnProtect();
}

/*--------------------------------------------------------------------------
```

```
 *    Main function
 *--------------------------------------------------------------------------
 */
void TaskMain ()
{
    XPEvent     em_stop_evt;    // the io-event message
                                // a derivate of a std. MCAT message
    MSGID       *xpevtid;       // msgid for subscription
    MSG         *msg;           // pointer to any type of message
    INT32       em_data;        // value of EMSTOP
    INTEGER     res;            // return code of WAITIO()
    UNSIGNED    len;            // length
    UNSIGNED    evtid;          // eventid

    // print banner.
    printf(" *** EDGE DEMO ***\n\n");

    // init ExpressIO
    if (!SYSTEM()) {
        TraceWriteLog("ExpressIo Init failed\n");
        TaskDelete(Self);
    } /* endif */

    // subscribe em_stop_evt, trigger on both edges
    xpevtid = XPEventSubscribe(&em_stop_evt,         // the event
                               1,                    // a user selectable event id
                               &em_stop,             // the IOOBJECT
                               IO_EVT_BOTH,          // event request mask
                               SYS_WAIT_INFINITE,    // timeout
                               &em_data,             // pointer to data
                               1);                   // size of em_data in INT32
    if (xpevtid == NULL) {
        printf("ERROR: Can't subscribe\n");
        TaskDelete(Self);
    } /* endif */

    // main loop
    loop {
        msg = MsgWait(0,SYS_WAIT_INFINITE);
        if (msg && msg->type == xpevtid->id) {
            // if we use msg here, we renew allways the correct
            // event
            XPEventRenew((XPEvent*)msg,&len,&evtid);
            // handle the events
            switch (evtid) {
              case 1:
                // EMSTOP
                if (len) {
                    printf("EMSTOP=%d\n",em_data);
                } else {
                    printf("ERROR: No data read\n");
                } /* endif */
                break;
```

```
             default:
               printf("ERROR: Unknown XPEvent\n");
            } /* endswitch */
        } else {
            printf("ERROR: Unknown MSG\n");
        } /* endif */
    } /* endloop */

    TaskDelete(Self);
}
```

3.2.3. Compile and Run

To run the example, it is necessary to connect the used digital out- and input. In this case we use the digital output 0 (marked **O1**) and the input (marked **I1**) of the CPU itself.

Now we can inspect the system using the ExpressIO commands of the SYSMON system monitor. First we use XLIST to get a list of installed modules and their ordinal numbers:

```
2+>xlist modules
BUS=CPU MODULE=01h TYPE=TSMARMCPU-DIN            CHANNELS=08
BUS=CPU MODULE=02h TYPE=TSMARMCPU-DOUT           CHANNELS=09
BUS=CPU MODULE=03h TYPE=TSMARMCPU-AIN            CHANNELS=08
BUS=CPU MODULE=04h TYPE=TSMARMCPU-AOUT           CHANNELS=02
BUS=CPU MODULE=06h TYPE=TSMARMCPU-EVTCNT         CHANNELS=02
BUS=CPU MODULE=07h TYPE=TSMARMCPU-FREQ           CHANNELS=08
BUS=TSM MODULE=00h TYPE=TSM-16A24P               CHANNELS=16  WATCHDOG
BUS=TSM MODULE=02h TYPE=TSM-16E24                CHANNELS=16
BUS=TSM MODULE=03h TYPE=TSM-4INC                 CHANNELS=04 POWERFAIL
BUS=TSM MODULE=04h TYPE=TSM-4DA16                CHANNELS=04 POWERFAIL
BUS=TSM MODULE=05h TYPE=TSM-8AD8-KTY             CHANNELS=08
BUS=TSM MODULE=07h TYPE=TSM-8AD12                CHANNELS=08
12 found.
```

From the output we learned that the digital inputs of the CPU have the module address 1. The output module has the address 2. Lets try if we connected them correctly:

```
2+>xin cpu.1.0
0
2+>xout cpu.2.0 1
2+>xin cpu.1.0
1
```

Now we look for installed ExpressIO *objects.* On a naked system there are none, so:

```
2+>xlist objects
0 found.
```

This is the expected behavior. Assuming we already loaded ***edge.shx*** into the system, we can start the demo now:

```
2+>init 402000
 *** EDGE DEMO ***
```

Because we created a named ExpressIO object, ***xlist objects*** should display something now.

```
2+>xlist objects
EMERGENCYSTOP
1 found.
```

Even better, we can use the name of the object to access the input. Note that object names are case sensitive.

```
2+>xin EMERGENCYSTOP
1
2+>xin emergencystop

FAIL
```

Well, we can switch the output now. Because we are looking for for edges only, our program should react:

```
2+>xout cpu.2.0  0
2+>EMSTOP=0
2+>xout cpu.2.0  1
2+>EMSTOP=1
```

Great! To terminate the session, the reset command is the choice:

```
2+>reset
```

You may play with the example. Try **IO_EVT_ONE** instead of **IO_EVT_BOTH**, for example.

4. What to Read Next?

Now its time to study the kernel reference, where the concepts and functions of the kernel is documented in detail. You may also have a look into the SYSMON or ExpressIO documentation.

III. SYSMON – A System Monitor

1. Introduction

SYSMOM is a *monitor program*. It can be used to retrieve system information like task status or interrupt status. It can also be used to download program images into systems RAM or FLASH memory.

2. Line Setup and Terminal Features

SYSMON is available via serial RS232 communication. The line setup is 19200-8-N-1 (19200 Baud, 8 bits per character, no parity, one stop bit). Usually the serial line used is marked “SER0” or “COM1”, depending on the naming convention of the hardware manufacturer. No hardware or software handshake is used.

SYSMON offers a very basic flow control when it comes to downloads. This feature is not documented, but mocom's own ***wLGO*** Terminal program supports this flow control. If you use a third party program (like *HYERTERMINAL*), you still can download SHX files. Use ASCII transfers and set the so-called line-pacing to greater than 1ms and less than 10ms.

SYSMON uses the ASCII control characters **BS** (*backspace, 0x08*) for line editing only. No cursor positioning codes are used. So line editing should work with almost all Terminal emulations. Important: Disable auto-echo of your terminal program!

There are a few ASCII control characters supported by SYSMON for the users convenience.

ASCII	***Comment***
ESC	Clear input. The current line is cleared, cursor is set to position 0.
CTRL B	Move cursor one position right, if possible.
CTRL F	Move cursor one position left, if possible.
CTRL K	Clear to end of line: Delete all characters right of the cursor. Cursor position is not moved.
CTRL A	Move cursor to begining of line (position 0).
CTRL E	Move cursor to end of line (behind last character).
CTRL W	Recalls the previous command line of the 4 line buffers available.
CTRL X	Recalls the next command line of the 4 line buffers available.
CTRL D	Delete character at the current cursor position.
CTRL H, **BACKSPACE**	Delete character in front of current cursor position (*rubout*).

ASCII	***Comment***
CTRL V	Toggle between override and insert mode. In insert mode, if a character is right of the cursor position, a newly entered character is inserted in front of this character. In override mode, if a character is right of the current cursor position, a newly entered character overwrights this character.

Using ***wlgo.exe*** you have not to deal with those codes, because *wlgo* maps the usual editing keys (*del,arrows,ESC*) to those control characters.

3. Basic Syntax

3.1. SYSMON is Case Sensitive!

SYSMON syntax is case sensitive! All commands are expected to be entered in lower case.

3.2. Prompt

The Sysmon prompt is ***2+>***

The 2 signals mCAT Version of at least 2.00. The + signals support for a faster flow control handling (used with wlgo only).

3.3. Numbers

Type	***Prefix***	***Example***
hexadecimal	a digit	67778, 0ffff, 6adcf ***(must start with digit, at least with a 0!)***
decimal	#	#10 #0 #-234
binary	%	%0101010101010101010

The default type is hexadecimal.

3.4. Strings

For some commands, the arguments have to be passed as strings. A string is enclosed by paragraphs. Example “huhu”. A paragraph within a string must be prefixed with a backslash. Exampel “This is the string \”huhu\””.

4. Command Reference

4.1. SYSMON Help

help – online help

Syntax: help {command}

Description: Display the help information. Without an argument, help gives a list of commands available. If the name of a specific command is given as an argument, detailed help on the specific command is shown.

Remarks:

Example:

```
2+>help
ps                 is                 msgids             msgid
libs               modules            mods               show
idle               mem                heap               pools
info               help               reset              dump
find               fill               move               crc
eemove             in                 out                S30..
upload             init               go                 suspend
resume             kill               eeread             eewrite
flashid            id                 erase              delpage
purge              val                blank              rmsys
settime            gettime            xin                xvin
xout               xvout              xinfo              xcfg
xlist              ipset              ips                format
dir                del                attrib             create
+                  -                  .                  *
:
Try help <command> for more information on specific commands.
all numbers are presumed to be in hex notation. Use a leading
'#' to indicate decimal notation. Use a leading '%' to indicate
binary notation (01010101). Hex numbers must not start with a letter
(A-F, a-f). In such cases use a leading '0'.

2+>help fill
fill <from> <to> {byte|word|long} <value>
 fill memory {byte|word|long} from memory
 location <from to <to> using <value>

2+>fill 402000 412000 long 044332211
2+>dump 411fc0
00411FC0 11 22 33 44 11 22 33 44 11 22 33 44 11 22 33 44 * ."3D."3D."3D."3D
00411FD0 11 22 33 44 11 22 33 44 11 22 33 44 11 22 33 44 * ."3D."3D."3D."3D
00411FE0 11 22 33 44 11 22 33 44 11 22 33 44 11 22 33 44 * ."3D."3D."3D."3D
00411FF0 11 22 33 44 11 22 33 44 11 22 33 44 11 22 33 44 * ."3D."3D."3D."3D
```

```
00412000 FF FF E0 D7 FF DF 08 36 BF FF 75 DE FF F7 20 A6 * .......6..u... .
00412010 FF FF 73 9F FF FF 02 EA FF FF 42 85 DF FF BD DE * ..s.......B.....
00412020 0C 22 D3 56 41 83 51 DD 02 02 04 B4 54 02 80 32 * .".VA.Q.....T..2
00412030 02 00 08 46 00 90 22 E6 22 20 27 59 20 00 E4 B6 * ...F..".." 'Y ...
2+>
```

4.2. mCAT Base Commands

is – interrupt status

Syntax: is

Description: A list of all installed ***interrupt drivers*** in the system is displayed.

Int# = the interrupt number or id
drv name = name of the driver module serving this interrupt
ver = version of the module as stored in the IMD
ws = address of the Interrupts workspace
p = interrupt priority
t = if 1, its a PASS_MSG style interrupt driver, o if it is an old style driver.
Int name = name of the interrupt source

Example:

```
2+>is
 int# | drv name          | ver  | ws        | p | t | int name
------+-------------------+------+-----------+---+---+--------------------
 0075 | mCAT/Ticker       | 3.02 | 045EFCB8h | 1 | x |INT_TIMER_0
 0077 | mCAT/BitbusDrv    | 1.00 | 045EDEFCh | 1 | x |INT_HDLC_RX_0
 0081 | mCAT/ETH0         | 1.00 | 045EB7F8h | 1 | x |INT_ETH_DMA_RX
 0082 | mCAT/ETH0         | 1.00 | 045EB858h | 1 | x |INT_ETH_MAC_TX
------+-------------------+------+-----------+---+---+--------------------
```

ps – program status

Syntax: ps {-t}

Description: A list of all running ***tasks*** in the system is displayed. If the option *-t* is given, a list of all active ***threads*** is given instead.

Remarks: The option *-t* is available on mCAT 2.20 ARM platforms only.

Example ***ps***:

```
 task | name             | vers | prio | thread    | qid | state
------+------------------+------+------+-----------+-----+------------------
 0000 | mcat/GBS         | 3.50 |  128 | 005ED478h | 017 | WAITING
 0011 | mCAT/HTTPD       | 1.10 |  128 | 0057C9C8h | 021 | WAITING
 0012 | mCAT/HTTPD       | 1.10 |  128 | 005879B0h | 020 | WAITING
 0013 | mCAT/IP          | 1.00 |  120 | 005C582Ch | 019 | WAITING
 0014 | mCAT/Sysmon      | 1.70 |  002 | 005EC634h | 018 | ACTIV
 0015 | mCAT/XPSERVER    | 1.01 |  254 | 005EF910h | 016 | WAITING
------+------------------+------+------+-----------+-----+------------------
```

task = task number

name = name of the task as found in the IMD

vers = version of the task as found in the IMD

prio = current priority

thread = base address of the main thread's structure

qid = the tasks queue id

state = the state of the task (see ***ThreadGetState***() function for details)

Example ***ps -t***:

```
 tid  | task | prio | addr      | sque   | stack | inuse | pc        | state
------+------+------+-----------+--------+-------+-------+-----------+---------
 0016 |  015 |  254 | 005EF910h | 000000 | 01024 | 00236 | 00BC0A20h | WAIT
 0017 |  000 |  128 | 005ED478h | 000000 | 01024 | 00252 | 00BC0A20h | WAIT
 0018 |  014 |  001 | 005EC634h | 000001 | 01024 | 00744 | 00BC0A20h | ACTIVE
 0019 |  013 |  120 | 005C4FD8h | 000000 | 04096 | 00180 | 00BC0A20h | WAIT
 0020 |  012 |  128 | 0058715Ch | 000000 | 08192 | 00180 | 00BC0A20h | WAIT
 0021 |  011 |  128 | 0057C170h | 000000 | 08192 | 00180 | 00BC0A20h | WAIT
 0022 | KRNL |  255 | 0055E068h | 000000 | 01024 | 00024 | 00BC0E78h | WAIT
------+------+------+-----------+--------+-------+-------+-----------+---------
```

Tid = thread id

task = number of the task the thread belongs to

prio = priority of the thread

sque = signal queue

stack = allocated stack size for this thread

inuse = currently used stack size (this is not the minimum! Do not try to set the stack size to this value!)

pc = the current program counter value. This value is equal for all threads in the state **WAIT**

state = the state of the task (see ***ThreadGetState***() function for details)

suspend – suspend a task

Syntax: suspend <taskno>

Description: Suspend task <taskno> (and all of its threads) from execution

Remarks:

resume – resume a task

Syntax: resume <taskno>

Description: Resume a previously suspended task <taskno> (and all of its threads).

Remarks:

kill – kill a task

Syntax: kill <taskno>

Description: Stop and remove a task (and its threads).

Remarks:

msgids – message ids

Syntax: msgids

Description: A list of all registered message ids in the system is displayed.

Type = the binary value used to identify a message, stored in the member ***type*** of an mCAT message.

User = the task that serves messages of the given type

Pool = you can attach a message pool id to a **MSGID** data structure. See also SYSMON command ***pools***.

Name = the registered name of the message id.

Remarks: In earlier versions of sysmon this command was named “msgid”. The old syntax is still available for your convenience.

Example:

```
2+>msgids
   type    | user  | pool  | name
-----------+-------+-------+--------------------
 00000B02h | 000Bh | 0000h | VIEW.cmd
 00000B01h | 000Bh | 0000h | mCAT/httpd/VIEW
 00000C02h | 000Ch | 0000h | SERVICE.cmd
 00000C01h | 000Ch | 0000h | mCAT/httpd/SERVICE
 00000D04h | 000Dh | 0000h | mCAT/IP/DNS
 00000D03h | 000Dh | FFFDh | mCAT/IP/SOCKET
 00000D02h | 000Dh | 0000h | mCAT/IP/IP
 00000D01h | 000Dh | 0000h | mCAT/IP/ARP
 00005202h | 0052h | FFFEh | mCAT/IPIF/SEND::eth0
 00005201h | 0052h | FFFEh | mCAT/IPIF/CFG::eth0
 00004601h | 0046h | 0000h | mCAT/COM2/Write
 00004701h | 0047h | 0000h | mCAT/COM2/Read
 00004401h | 0044h | 0000h | mCAT/COM1/Write
 00004501h | 0045h | 0000h | mCAT/COM1/Read
 00000E01h | 000Eh | 0000h | mCAT/SYS/SERVICE/cmd
 00000001h | 0000h | 0000h | mCAT/GBS
 00004D01h | 004Dh | FFFFh | mCAT/Bitbus
 00000F02h | 000Fh | 0000h | mCAT/XPSERV/IoServer
 00000F01h | 000Fh | 0000h | mCAT/XPSERV/Subscribe
 00004B02h | 004Bh | 0000h | mCAT/Ticker
 00004B01h | 004Bh | 0000h | mCAT/Timer
-----------+-------+-------+--------------------
```

libs – list shared libraries

Syntax: libs

Description: A list of all registered shared libraries in the system is displayed.

Remarks:

```
2+>libs
 libid | name             | ver
-------+------------------+------
 0000h | mCAT/Kernel      | 3.00
 0001h | mCAT/MemMgr      | 3.01
 0002h | mCAT/NameServer  | 2.00
 0003h | mCAT/NvMemMgr    | 1.10
 0005h | mCAT/GBSAPI      | 1.10
 0008h | mCAT/IP/socket   | 3.00
 0009h | mCAT/ExpressIO   | 2.10
 000Ah | mCAT/Ticker      | 3.02
 000Bh | mCAT/FlashMaint  | 2.00
 000Ch | mCAT/XPIO/Server | 1.10
 0010h | mCAT/SimpleIO    | 1.30
 0016h | mCAT/BGMAPI      | 1.02
 0018h | mCAT/HTTPDLib    | 1.00
-------+------------------+------
```

modules – list of all modules in FLASH memory

Syntax: modules {-m}

Description: A list of all start-able modules (tasks, interrupt drivers, shared libraries, ..) stored in the systems FLASH memory is displayed. If the option ***-m*** is used, the used and reserved memory space for those modules are displayed, separated for ROM (=FLASH) and RAM. If a module does not support the new style IMD introduced with the mCAT Version 2.20 development environment, no memory layout data is displayed. In the example following, the module “mCAT/TSM/BOOT” does so.

Remarks: In earlier versions of sysmon this command was named “mods”. The old syntax is still available for your convenience.

Example:

```
2+>modules -m
```

```
 name              | ver  | rom.base  | rom.len   | ram.base  | ram.len
------------------+------+-----------+-----------+-----------+-----------
 mCAT/IdleTask     | 1.00 | 00BC0000h | 00008B30h | 007F0000h | 00007EBCh
 mCAT/Kernel       | 3.00 | 00BC0000h | 00008B30h | 007F0000h | 00007EBCh
 mCAT/NvMemMgr     | 1.10 | 00BC8B38h | 00001200h | 007F7EC0h | 00000018h
 mCAT/MemMgr       | 3.01 | 00BC9D40h | 00000EE8h | 007F7EE0h | 000004A8h
 mCAT/NameServer   | 2.00 | 00BCAC30h | 00000A50h | 007F8390h | 00000000h
 mCAT/Ticker       | 3.02 | 00BCB688h | 00000D90h | 007F8398h | 00000010h
 mCAT/ExpressIO    | 2.10 | 00BCC420h | 000016BCh | 007F83B0h | 00000010h
 mCAT/XPIO/Server  | 1.10 | 00BCDAE0h | 000024F0h | 007F83C8h | 00000020h
 mCAT/XPSERVER     | 1.01 | 00BCDAE0h | 000024F0h | 007F83C8h | 00000020h
 mCAT/FlashMaint   | 2.00 | 00BCFFD8h | 00000B94h | 007F83F0h | 00000000h
 mCAT/BitbusDrv    | 1.00 | 00BD0B70h | 000026B8h | 007F83F8h | 00000048h
 mcat/GBS          | 3.60 | 00BD3230h | 0000548Ch | 007F8448h | 000000C8h
 mCAT/GBSAPI       | 1.10 | 00BD3230h | 0000548Ch | 007F8448h | 000000C8h
 mCAT/TSM/ARMCPU   | 1.02 | 00BD86C0h | 00001CACh | 007F8518h | 00000048h
 mCAT/TSM/DINOUT   | 2.01 | 00BDA370h | 000006C0h | 007F8568h | 00000000h
 mCAT/TSM/2da12    | 1.11 | 00BDAA38h | 00000784h | 007F8570h | 00000048h
 mCAT/TSM/4DA16    | 1.01 | 00BDB1C0h | 00000860h | 007F85C0h | 00000048h
 mCAT/TSM/8ad12    | 2.00 | 00BDBA28h | 00000D40h | 007F8610h | 00000048h
 mCAT/TSM/8ad8     | 1.00 | 00BDC770h | 00000CC0h | 007F8660h | 00000048h
 mCAT/TSM/1inc     | 1.00 | 00BDD438h | 0000037Ch | 007F86B0h | 00000000h
 mCAT/TSM/4INC     | 1.00 | 00BDD7B8h | 00000680h | 007F86B8h | 00000000h
 mCAT/SWBUS/float  | 1.00 | 00BDDE40h | 00000784h | 007F86C0h | 00000000h
 mCAT/SWBUS/int    | 1.00 | 00BDE5C8h | 00000788h | 007F86C8h | 00000000h
 mCAT/Sysmon       | 1.70 | 00BDED58h | 0000DE70h | 007F86D0h | 00000D0Ch
 mCAT/httpdfs      | 1.00 | 00000000h | 00000000h | 00000000h | 00000000h
 mCAT/SerDrv       | 2.03 | 00B48000h | 0000487Eh | 00401A00h | 00000000h
 mCAT/SimpleIO     | 2.02 | 00B58000h | 00002756h | 00401B00h | 00000000h
 mCAT/BGMServer    | 1.04 | 00B80000h | 000078A4h | 00400000h | 00000244h
 mCAT/BGMAPI       | 1.02 | 00B878A8h | 00000DF8h | 00400248h | 00000000h
 mCAT/ETH0         | 1.00 | 00B886A8h | 00001B20h | 00400250h | 00000008h
 mCAT/IP           | 1.00 | 00B8A1D0h | 00007F48h | 00400260h | 00000A78h
 mCAT/IP/socket    | 3.00 | 00B8A1D0h | 00007F48h | 00400260h | 00000A78h
 mCAT/HTTPD        | 1.20 | 00B92120h | 0000F6A4h | 00400CE0h | 000004ACh
 mCAT/HTTPDLib     | 1.00 | 00B92120h | 0000F6A4h | 00400CE0h | 000004ACh
 mCAT/TSM/BOOT     | 1.00 | --------- | --------- | --------- | ---------
------------------+------+-----------+-----------+-----------+-----------
```

show – display the bootlog

Syntax: show

Description: While mCAT starts up all information printed using the TraceWriteLog() kernel function is stored in a ring buffer – called the bootlog. Using the command ***show*** you can display this log. If the system has many modules installed and/or uses TraceWriteLog frequently, the bootlog may not hold a complete boot protocol.

Example:

```
mCAT V2.20-R00168 TSMARMCPU [SAMSUNG S3C4530 ARM7TDMI]
```

```
(c) 1997-2004 mocom software Gmbh & Co KG
email: support@msac.de

mCAT/IdleTask 1.00
mCAT/Kernel 3.00
mCAT/NvMemMgr 1.10
mCAT/MemMgr 3.01
 2048 kByte RAM found
 256 kByte HEAP installed
mCAT/SimpleIO 1.30
mCAT/NameServer 2.00
mCAT/Ticker 3.02
mCAT/ExpressIO 2.10
mCAT/XPSERVER 1.01
mCAT/FlashMaint 2.00
mCAT/BitbusDrv 1.00
 NODE=2 SPEED=1.5 MBit/s BUFFERS=8 MSGLEN=255
mcat/GBS 3.50
mCAT/GBSAPI 1.10
mCAT/TSM/ARMCPU 1.01
mCAT/TSM/DINOUT 2.01
mCAT/TSM/4DA16 1.01
mCAT/TSM/8ad12 2.00
mCAT/TSM/8ad8 1.00
mCAT/SWBUS/float 1.00
mCAT/SWBUS/int 1.00
mCAT/Sysmon 1.70
mCAT/httpdfs 1.00
mCAT/BGMServer 1.04
 BGM: NO NVRAM AREA FOUND, BGM NOT INSTALLED
mCAT/BGMAPI 1.02
mCAT/ETH0 1.00
 ETH MAC: 00.06.EA.00.00.0C
 ETH 100-BASE-TX, HALF DUPLEX
 ETH MTU=1500, BUFFERS=100
 INTERFACE IP = 172.031.031.053 [AC1F1F35]
 GATEWAY ADDR = 172.031.031.253 [AC1F1FFD]
mCAT/IP 1.00
 USING DOMAIN NAMESERVER AT 172.031.031.254 [AC1F1FFE]
mCAT/IP/socket 3.00
mCAT/HTTPD 1.10
mCAT/HTTPDLib 1.00
mCAT/TSM/BOOT 1.00
SYSTEM STARTED

 *** SYSMON 1.70 ***


2+>
```

mem – display the memory utilization

Syntax: mem

Description: Display the size and utilization of installed RAM, the system heap and the so-called NVRAM section (used by BGMEM to store data non-volatile).

Remarks:

heap – display the heap utilization

Syntax: Heap {*module name*}

Description: Display base address, length of memory block and owner (imd name) of all allocated blocks of memory. If *module name* is given, only blocks allocated for the given module name are shown.

Example:

```
2+>heap mcat/gbs
 #     | base      | size     | module
-------+-----------+----------+------------------
 00001 | 005ec81ch | 00000048 | mcat/GBS
 00002 | 005ec84ch | 00000076 | mcat/GBS
 00003 | 005ec898h | 00002056 | mcat/GBS
 00004 | 005ed0a0h | 00000164 | mcat/GBS
 00005 | 005ed144h | 00000076 | mcat/GBS
 00006 | 005ed190h | 00000032 | mcat/GBS
-------+-----------+----------+------------------
```

pools – display the buffer pool utilization

Syntax: pools

Description: Display the size and utilization of installed buffer pools. The information includes number of buffers in a pool, number of free buffers in a pool and the size of a single buffer.

Start and *end* mark the memory region from where the buffers of a given pool are allocated. Len gives the length of the buffers in a specific pool. Buffers tells the number of buffers a pool was designed to hold. Free tells how many buffers are still available. Owner tells the thread id or interrupt id of the creator of that pool.

Pools is a powerful help when it come to debugging. If a applictaion or driver consume more buffers than available or constantly holds more buffers than expected, something is pretty wrong.

Remarks: This command is available on mCAT 2.20 ARM platforms only.

Please note that some drivers initially allocates a set of buffers for internal optimization. This is NOT A BUG!

Example:

```
2+>pools
 pool | start     | end       | len   | buffers | free  | owner
------+-----------+-----------+-------+---------+-------+-------
 000d | 005ED598h | 005EDEB8h | 00292 |   00008 | 00007 | 077
 001d | 005C50D0h | 005EAF10h | 01552 |   00100 | 00060 | 081
 002d | 005B7CF4h | 005BF0F4h | 00116 |   00256 | 00256 | 013
 003d | 005874F4h | 005B7CF4h | 01552 |   00128 | 00128 | 013
------+-----------+-----------+-------+---------+-------+-------
```

info – display system information

Syntax: info

Description: Display the hardware serial number, the build date and version of the mCAT kernel and some other information of interest (including BIT-BUS and ETHERNET settings).

Remarks: In earlier versions of SYSMON this command was named “serno”. The old syntax is still available for your convenience.

reset – issue a system reset

Syntax: reset

Description: This command disables all interrupts and stops the servicing of the hardware watchdog. As a result, a hardware reset will be issued.

Remarks:

4.3. Memory and I/O Manipulation Commands

dump – display memory

Syntax: dump {addr}

Description: Display 128 bytes of memory starting with {addr} in hexadecimal format. A ASCII representation of the bytes is also shown where applicable .

Remarks:

find – find in memory

Syntax: find <startaddr> <endaddr> {!} <byte> <string> ...

Description: Find a sequence of bytes / string in memory. The search will start at <startaddr> and end at <endaddr>. A single exclamation mark (!) after <endaddr> forces a non-case-sensitive comparison.

Remarks:

fill – fill memory

Syntax: fill <startaddr> <endaddr> {byte|word|long} <value>

Description: Fill memory from <startaddr> to <endaddr> with<value>. By default, value is assumed to be a byte value. Use the modifiers ***byte***, ***word*** or ***long*** to specify the width of value.

Remarks:

move – move memory

Syntax: move <from> <to> <length>

Description: Move a chunk of memory from address <from> to address <to>. The length of the chunk is <length>.

Remarks:

blank – blank check memory

Syntax: blank <from> <to>

Description: Checks whether the given memory area is blank (all 0xFF). Operation starts at address <from> and ends at address <to>.

Remarks:

crc – calculate the CRC32 code for a memory region

Syntax: crc <from> <to>

Description: Calculate the CRC32 *cyclic redundancy check* code for the memory area from address <from> to address <to>.

Remarks: This function can be helpful to detect changes in regions where no changes are expected.

in – read a byte, word or long word from the i/o space

Syntax: in {byte|word|long} <addr>

Description: Read a byte, word or long word from the i/o space. The value is displayed in hexadecimal, decimal and binary notation. If no width modifier is given, a byte is read.

Remarks: On the Toshiba TLCS900 and ARM implementations of mCAT i/o is memory mapped. This implies that you can use ***in*** to read values in memory as well. This feature may not be guaranteed on future platforms as they may have separate i/o spaces.

On ARM platforms you can cause an unaligned access exception (system will reset!) if you specify word or long word access and your address is not properly aligned. Example: ***in long 1***

out – write a byte, word or long word into the i/o space

Syntax: out {byte|word|long} <addr> <value>

Description: Write a byte, word or long word into the i/o space. No output other than a new line is made. If no width modifier is given, a byte is written.

Remarks: On the Toshiba TLCS900 and ARM implementations of mCAT i/o is memory mapped. This implies that you can use ***out*** to write a values to memory as well. This feature may not be guaranteed on future platforms as they may have separate i/o spaces.

On ARM platforms you can cause an unaligned access exception (system will reset!) if you specify word or long word access and your address is not properly aligned. Example: ***out word 1 0aaff***

4.4. Flash Memory Manipulation Functions

flashid – read the device id from flash memory

Syntax: flashid <addr>

Description: Returns type of the installed Flash at address <addr>. If no <addr> is given, a list of supported Flash types is displayed.

Remarks: In earlier versions of sysmon this command was named “id”. The old syntax is still available for your convenience.

erase – erase flash memory

Syntax: erase <addr>

Description: Erase a sector of a FLASH-Memory starting at <addr>. This command immediately starts to physically erase the flash sector addressed by <addr>.

Remarks: If any type of program is still running from within such a page, the system WILL CRASH! This calls disables all interrupts until the operation is complete. This can take a few seconds and may harm the system functionality. To prevent those pitfalls use the command ***delpage*** instead.

delpage – mark flash pages for deletion

Syntax: delpage <pageno> {<pageno>...<pageno>}

Description: Erase flash using flash maintainer 'FERA'. A list of up to 16 ordinal page numbers is used to mark individual pages for deletion. The actual deletion process is started after system reset when no interrupts are active. This prevents the pitfalls of the ***erase*** command.

Example:

delpage 0 1 2 #12

Erase pages 0, 1, 2 and 12 after next system reset.

Remarks: *The base addresses of the pages and their associated page numbers are hardware depended and are documented in the mCAT release documentation!*

purge – invalidate a module

Syntax: purge <addr>

Description: Invalidate an IMD (AA55 => 0055) at <addr>. This command is useful to deactivate a module in FLASH memory. Every mCAT module starts with an IMD. The first entry in the IMD is the pattern 0xaa55. If this pattern has another value, the module is not detected and as a consequence not installed.

Remarks:

val – validate a module

Syntax: val <addr>

Description: Validate an IMD (FF55 => AA55) at <addr>. This command is useful to activate a module in FLASH memory. Every mCAT module starts with an IMD. The first entry in the IMD is the pattern 0xaa55. If this pattern was set to 0xff55 instead at compile time, the module is not detected and as a consequence not installed. However, you can still use the SYSMON command ***init*** to start such a module. After use of ***val*** the pattern will be changed and the module will be detected with the next reboot.

Remarks: The commands val and purge are useful when you move a module from RAM to FLASH. It is a good practice to set the IMD to 0xff55before you move to FLASH. Then you can use ***init*** to further test the modules behavior in the FLASH environment. When you are sure its working even in FLASH, use ***val*** to validate it and reboot the system. The module will then be started automatically by the system.

4.5. EEPROM Manipulation Functions

eeread – read eeprom

Syntax: eeread <addr>

Description: Read one 16-Bit word from the serial EEPROM.

Remarks:

eewrite – write eeprom

Syntax: eewrite <addr> <value>

Description: Write one 16-Bit word into the serial EEPROM.

Remarks:

eemove – move in eeprom memory

Syntax: eemove <from> <to> <length>

Description: Move a chunk of memory in the serial eeprom from address <from> to address <to>. The length of the chunk is <length>. Please note that all addresses are 16-Bit values.

Remarks:

4.6. Program Upload / Download and Start

S3.. – store a S3/S7 format hex record in memory

Syntax: A valid S3/S7 record

Description: Every valid S3 format record send to SYSMON is accepted. The data portion of the record is written to the memory using the embedded address. SYSMON detects whether the target memory is RAM or FLASH and it will invoke the FLASH programming functions as needed.

Remarks:

upload – memory upload

Syntax: upload <from> <to> {<recsize>}

Description: Send the memory region from address <from> to address <to> to the the display using S3/S7 hex file format. If you activate logging with ***wlgo.exe***, you can retrieve the S3/S/ data from the logfile later.

Remarks: The record size is 64-bytes per record per default. You can modify the record size using the optional argument <recsize>.

rmsys – remove system

Syntax: rmsys

Description: If you have to replace the current mCAT by another version, use ***rmsys*** invalidate the mCAT core module, to mark all flash pages that need to be deleted and to issue a reset. After the reset, the so-called BOOTMON software (looks pretty like std. MCAT) will allow you to download a new mCAT S3/S7 image.

Remarks: Useful for system updates.

init – init module

Syntax: init <addr>

Description: 'init' a module (e.g. create a task) by executing its TaskInit() function. Useful to start modules in RAM or disabled modules (see command purge) in FLASH memory.

Remarks:

go – call function

Syntax: go <addr>

Description: Execute code at <addr>. The code is executed in the task space of SYSMON so you can easily hang up SYSMON! Be careful!

Remarks: Used for very special situations only!

4.7. Optional Commands

4.7.1. ExpressIO Specific Commands

SYSMON supports a minimum set of commands to explore and test ExpressIO. It allows both, access to physical drivers via ***bus.module.channel*** and access to named IOOBEJCTS.

xlist – list ExpressIo properties

Syntax: Xlist modules|xp|objects|busses

Description: Xlist can be used to explore the installed ExpressIo. Using the sub-options, it is easy to verify the installed hardware and the available ExpressPrograms.

modules = list all hardware modules installed

xp = list all ExpressPrograms available

objects = list all named ExpressObjects in the system

busses = list physically available buses

Remarks:

Example output:

```
2+>xlist xp
CAPTURE
CAPTURE
VECCOUNTER
PULSE
EDGE
COUNTER
6 found.
2+>xlist xp busses
CPU   SLOTS=8 MODULES INSTALLED=6
TSM   SLOTS=16 MODULES INSTALLED=6
SFT   SLOTS=64 MODULES INSTALLED=0
3 found.
2+>
xlist modules
BUS=CPU MODULE=01h TYPE=TSMARMCPU-DIN            CHANNELS=08 POWERFAIL
BUS=CPU MODULE=02h TYPE=TSMARMCPU-DOUT           CHANNELS=09
BUS=CPU MODULE=03h TYPE=TSMARMCPU-AIN            CHANNELS=08
BUS=CPU MODULE=04h TYPE=TSMARMCPU-AOUT           CHANNELS=02
BUS=CPU MODULE=06h TYPE=TSMARMCPU-EVTCNT         CHANNELS=02
BUS=CPU MODULE=07h TYPE=TSMARMCPU-FREQ           CHANNELS=08
BUS=TSM MODULE=00h TYPE=TSM-16A24P               CHANNELS=16  WATCHDOG
BUS=TSM MODULE=02h TYPE=TSM-16E24                CHANNELS=16
BUS=TSM MODULE=03h TYPE=TSM-4INC                 CHANNELS=04 POWERFAIL
BUS=TSM MODULE=04h TYPE=TSM-4DA16                CHANNELS=04 POWERFAIL
BUS=TSM MODULE=05h TYPE=TSM-8AD8-KTY             CHANNELS=08
BUS=TSM MODULE=07h TYPE=TSM-8AD12                CHANNELS=08
12 found.
```

xin – read a single channel

Syntax: xin <bus>.<module>.<channel> | <ioobject>

Description: Read and display the value of a single channel. The channel can be addressed using the *bus.module.channel* enumeration scheme or via existing ExpressObjects.

Remarks:

Example 1 (*cpu.1.1* is a digital input):

```
2+> xin cpu.1.1
0
2+>
```

Example 2 (cpu.3.1 is an analog input):

```
2+> xin cpu.3.1
000003F9h [1017]
2+>
```

Example 3 (a named IOOBJECT, a digital input)

```
2+> xin belt_moves
0
2+>
```

xout – write to a single channel

Syntax: xout <bus>.<module>.<channel> | <ioobject> <value>

Description: Write the value <*value*> to a specific channel. The channel can be addressed using the *bus.module.channel* enumeration scheme or via existing ExpressObjects.

Remarks:

Example output:

```
2+> xout cpu.2.8 1
2+>
```

xvin – read all channels of a module

Syntax: xin <bus>.<module> | <ioobject>

Description: Read all channels of a module. The module can be addressed using the *bus.module.channel* enumeration scheme or via existing ExpressObjects.

Remarks:

Example 1 (digital i/o):

```
2+> xvin cpu.2
0000.0000.0
2+>
```

Example 2 (analog i/o):

```
2+>xvin cpu.2 3
00000432h [1074]
00000437h [1079]
0000045Ah [1114]
00000469h [1129]
0000045Ah [1114]
00000459h [1113]
00000468h [1128]
00000001h [1]
2+>
```

xvout – write to all channels of a module

Syntax: xvout <bus>.<module> | <ioobject> <val0> <val1> .. <valN>

Description: Write a list of values (<val0> <val1> .. <valN>) to the channels of a module. The values will be assigned to the outputs as they appear in the list (*<val0>* == *channel 0*, *<val1>* == *channel 1 ...*).

Remarks:

Example:

```
2+> xvout cpu.2 1 1 0 0 1 1 0 0 1
2+>
```

xinfo – read a configuration item

Syntax: xinfo <bus>.<module>{.<channel>} | <ioobject> <item>

Description: Read a configuration item. The item identifier <*item*> must be a numeric constant. If a module specific item shall be read, the argument <channel> is optional.

Remarks: See also 4.7.1.1. Constant Values Used for xinfo & xcfg Commands

Example output:

```
2+> xinfo cpu.1.0 2
TSMARMCPU-DIN
2+> xinfo cpu.1 2
TSMARMCPU-DIN
2+>
```

xcfg – write a configuration item

Syntax: xcfg <bus>.<module>.<channel> | <ioobject> <item> <value>

Description: Write a configuration item. The item identifier <*item*> and the value <*value*> must be numeric constants. If a module specific item shall be read, the argument <channel> is optional.

Remarks: See also 4.7.1.1. Constant Values Used for xinfo & xcfg Commands

Example output (set analog input channel range to 0-5V):

```
2+> xcfg cpu.3.1 #10 012
```

4.7.1.1. Constant Values Used for xinfo & xcfg Commands

This section lists all constant values needed for the xcfg & xinfo commands in SYSMON compatible form. For details on items, the use and meaning of the cfg/info calls and those constants, please refer to the ExpressIO documentation.

```
// XCFG items
CFG_SET_ENABLE            6
CFG_SET_CHANNEL_RANGE     #10
CFG_SET_CONV_SPEED        #11
CFG_SET_GAIN              #12
CFG_SET_ATTENTUATE        #13
CFG_SET_OFFSET            #14
CFG_SET_LINTAB            #15
CFG_SET_PWM_FREQ          #16
CFG_SET_INC_MODE          #20
```

```
CFG_SET_DIR                 #21
CFG_SSI_SET_TURNS           #22
CFG_SSI_SET_STEPS           #23
CFG_SET_CHANNELS            #30
XP_CFG_SET_SAMPLE_RATE      08001
XP_CFG_SET_LOWTIME          08002
XP_CFG_SET_HIGHTIME         08003
XP_CFG_SET_TRIGGERMODE      08004
XP_CFG_SET_INVERSION        08005
// XINFO items
INFO_GET_IDENT              1
INFO_GET_IDENT_STRING       2
INFO_GET_VECTOR_SIZE        3
INFO_GET_PORT_CLASS         4
INFO_GET_POWERFAIL          5
INFO_GET_WATCHDOG           6
INFO_GET_CHANNEL_RANGE      7
INFO_GET_START_MODE         8
INFO_GET_GAIN               9
INFO_GET_ATTENTUATE         #10
INFO_GET_OFFSET             #11
INFO_GET_LINTAB             #12
INFO_GET_ADDRESS            #13
INFO_GET_INC_MODE           #14
INFO_GET_INDEX_STATUS       #15
INFO_GET_CHANNELS           #30
XP_INFO_GET_IDENT_STRING    08000
XP_INFO_GET_SAMPLE_RATE     08001
XP_INFO_GET_MAX_SAMPLE_RATE 08002
XP_INFO_GET_LOWTIME         08004
XP_INFO_GET_HIGHTIME        08005
XP_INFO_GET_TRIGGERMODE     08006
XP_INFO_GET_INVERSION       08007
// I/O CLASS INFO
CLASS_DIGITAL               00001
CLASS_ANALOG                00002
CLASS_PWM                   00004
CLASS_FREQ                  00008
CLASS_EVTCNT                00010
CLASS_POS                   00020
CLASS_INTEGER               00040
CLASS_FLOAT                 00080
CLASS_INPUT                 08000
CLASS_OUTPUT                00000
CLASS_RMW                   04000
// AD/DA CONFIGURATION VALUES
RANGE_RAW                   001
RANGE_RAW_U_10000           002
RANGE_RAW_U_5000            003
RANGE_RAW_S_10000           004
RANGE_RAW_S_5000            005
RANGE_U_10000               011
RANGE_U_5000                012
```

```
RANGE_S_10000               014
RANGE_S_5000                015
RANGE_UB                    01f
RANGE_020mA                 020
RANGE_420mA                 021
RANGE_PT100V4               031
RANGE_KTY                   032
RANGE_KTY10                 032
RANGE_KTY81                 032
RANGE_LM34                  034
RANGE_THERMO_K              041
RANGE_THERMO_J              042
// TSM4SSI ONLY
SSI_START_0                 01
SSI_START_1                 02
SSI_START_2                 04
SSI_START_3                 08
SSI_START_ALL               0f
// COUNTER MODE FOR TSM4INC
INC_MODE_QUADRATURE_X1      1
INC_MODE_QUADRATURE_X2      2
INC_MODE_QUADRATURE_X4      3
// MODE FOR PULSE EXPRESSPROGRAM
TRG_SINGLE                  1
TRG_RETRIGGER               2
TRG_QUEUED                  3
INVERT_OFF                  0
INVERT_ON                   1
```

4.7.2. Realtime Clock Specific Commands

settime – set system RTC

Syntax: settime #hh #mm #ss #dd #mm #yyyy

Description: Set the real-time clock. Be sure to use the '#' to signal SYSMON decimal input! Example:

settime #16 #30 #23 #24 #02 #2003

will set the RTC to:

24.02.2003 16:30:23

Remarks: Time must be in UTC!!

gettime – display system time (RTC)	
Syntax:	gettime
Description:	Print out current time [UTC]
Remarks:	Time must be in UTC!!

4.7.3. BGMEM Specific Commands

4.7.3.1. format {size}

Formats BgMem - deletes all files. If a size is specified and if this differs from EEPROM word 15 then this new size is written to EEPROM and a RESET is executed to make the new memory layout active. All data in the BgMem memory is lost.

4.7.3.2. dir

Shows a list of available files

```
2+>dir
---F EventFifo.log 00128 / 00040
     CREATED TUE 2001-7-10 8:3:59
     LAST MODIFIED TUE 2001-7-10 8:3:59
00001 good files found
```

The first letters represent the file attributes. The first two are reserved for extensions, the 3rd is “R” for Read only or “-“ for R/W. The last attribute is the file type “-“ for random file, “F” for FIFO, “L” for LIFO and “R” for ring buffer.

The file name, the number of records and their size follow.

4.7.3.3. attrib <filename> <+r|-r>

The first argument is the file name, the second a switch. “+r” activates read only.

4.7.3.4. del <filename>

Deletes a file.

4.7.3.5. create <filename><#records><#size><fifo|lifo|ring|random>

Creates a file <filename> with <#records> of <#size>each. The structure of the file is “fifo” or “lifo” or “ring” or “random”.

A size of 0 selects a file of maximum size (BGM_MODE_FIT)

4.7.4. SOCKET / Ethernet Specific Commands

4.7.4.1. IP-Information at a Glance: *info* and *ips*

The mCAT system monitor SYSMON offer a command called *info*. It takes no arguments and displays some system configuration values – including the IP configuartion. Simply issue *info* at the command line and you will get an output like:

```
mCAT     Version 2.20-D00166 Build 401FC857
         (c) 1997-2004 mocom software Gmbh & Co KG
         email: support@msac.de

         SERNO=DAV092
         SYSSTART=00BC0000 HARD=0012 CORE=1101

BITBUS   NODE=2 SPEED=1.5 MBit/s BUFFERS=8 MSGLEN=255
         RADIO MODE = OFF

ETHERNET 0:
         MAC=0006 EA00 000C
         AUTONEGOCIATE

         IP ADDR     = 172.031.031.053
         SUBNETMASK  = ---.---.---.---
         GATEWAY     = 172.031.031.253
         DNS         = 172.031.031.254
         MTU         = 1500
         MAX SOCKETS = 64

---.---.---.--- means “not set”
```

4.7.4.2. Setup IP-Addresses: *setip*

The *ipset* command can be used to setup the IP configuration of a system. At the command line the command expects an item selector and an address as an argument.

Please note that the address MUST be enclosed in quotes.

Please also note that the IP address is the only value that MUST be set to operate a mCAT node in a simple intranet.

```
ipset IP "xxx.xxx.xxx.xxx" set the IP address
ipset NM "xxx.xxx.xxx.xxx" set the NETMASK if other than default
ipset GW "xxx.xxx.xxx.xxx" set the GATEWAY address (optional)
ipset DN "xxx.xxx.xxx.xxx" set the DOMAIN NAME SERVER address
```

```
Example:
ipset ip "172.31.31.54"
```

Setting a GATEWAY address allows the node to communicate with hosts in other networks (e.g. The Internet). To prevent this set *GATEWAY* to “255.255.255.255” (will be displayed as “not set”). To prevent that a host from another network (e.g. the internet) connects itself to your local nodes, set *GATEWAY* to “0.0.0.0”. In that case, the mCAT TCP/UDP/IP protocol stack accepts only traffic from and to hosts attached to the same network (controlled by the network mask).

If you do not need a GATEWAY, it is recommended to set gateway to “0.0.0.0”.

4.7.4.3. List IP-Status: *ips*

This command lists all allocated sockets. Note that TCP server side anonymous sockets used to maintain connections have an `rport` (remote port) and an `rip` (remote ip address) ONLY!

```
 #   | prt | port  | rport | rip             | lis | acid  | endpoint
-----+-----+-------+-------+-----------------+-----+-------+------------------
 001 | TCP |  8000 |     0 | 000.000.000.000 | 001 |     0 | mCAT/httpd/SERVICE
 002 | TCP |    80 |     0 | 000.000.000.000 | 004 |     0 | mCAT/httpd/VIEW
-----+-----+-------+-------+-----------------+-----+-------+------------------
```

4.7.4.4. Set/Display Ethernet Mode

(New with R00409 and later)

This command can be used to set the Ethernet mode. The standard mode is AUTONEGOCIATE. On some systems the standard is AUTONEGOCIATE10 (limiting the negotiation to the 10MBit modes) to limit the power dissipation.

```
ethmode {<infno>} "<mode>"
 set/display phymode of ethernet interface <infno>.
 If <infno> is not given, std. interface 0 is used.
 <mode> can be of:
  auto,autu10,auto100,full100,full10,half100,half10,isolate
```

The command displays the current preset mode (the one that is stored in eeprom and that is and will be valid after startup) and the current state of the physical interface (PHY). If you use *ethmode, you* change the preset mode only.

```
2+>ethmode
```

```
 PRESET MODE  : AUTONEGOTIATION
 CURRENT STATE: 100-BASE-TX, HALF DUPLEX
2+>ethmode "100full"
 PRESET MODE  : 100-BASE-TX, FULL DUPLEX
 CURRENT STATE: 100-BASE-TX, HALF DUPLEX
2+>reset
      ....
2+>ethmode
 PRESET MODE  : 100-BASE-TX, FULL DUPLEX
 CURRENT STATE: 100-BASE-TX, FULL DUPLEX
```

4.7.4.5. Set the TELNETD Password

(New with R00409 and later)

Using the command *passwd* you a new password for the telnet connection can be set. Please note that only one TELNET connection can be opened to SYSMON at a time. The telnet connection disables the serial connection to Sysmon as well.

Do not try to download SHX-Files via telnet, because this protocol is not prepared to handle the necessary handshake.

```
2+>help passwd
passwd "<oldpasswd>" "<newpasswd>"
 change telnetd password. If no password is used up to now, a
 emtpy string "" must be used.
```

IV. mCAT Kernel Reference

1. The mCAT Kernel Technical Reference

The mCAT-Kernel is designed to control and manage the resources of the central processing unit (CPU). The main resource of the CPU is the *processing time* – and that is the focus of the kernel functions. Other important resources, like memory resources, are managed in separate modules – and this is again another important feature of the basic mCAT concept: A functionality can be implemented in separated *modules*. To achieve the modularity needed therefore, mCAT offers different kinds of modules:

1. Tasks / Threads
2. Interrupt-Drivers
3. Shared Libraries
4. Inits

The functions needed to manage and to interconnect these modules are incorporated in the mCAT-Kernel. The glue needed to interconnect all modules is the *message passing.*

1.1. Typographic Conventions

A few conventions should help us to keep the overview ...

A word or phrase of importance is set in *italic* style.

A Constant defined in a header file included is set in **this** style.

1.2. A note on Datatypes

With mCAT 2.10-R00168 we have changed a few data types and names of data types for better compatibility. However, the old mCAT 2.10 data types are still available and fully valid. Please take the following table as a reference for the different types:

MCAT 2.10 and earlier	***MCAT 2.10-R00168***	***minimum***	***maximum***	***use***
byte	UINT8	0	255	
word	UINT16	0	65535	
lword	UINT32	0	4294967295	
-	UINT64	0	2^{64}-1	Not available with TLCS900
-	INT8	-128	127	
short	INT16	-32768	32767	
long	INT32	-2147483648	2147483647	
-	INT64	-2^{63}	2^{63}-1	Not available with TLCS900
int*	INTEGER*	-32768*	32767*	Default type for small integers
unsigned*	UNSIGNED*	0	65535*	Default type for unsigned integers
bool	BOOL	TRUE	FALSE	

* The range of those types depend on the target platform. We assume that for those types the *16-Bit* data range is the minimum we can rely on.

2. mCAT Tasks

2.1. The Concept of Tasks

This section describes task related functions. A task is the smallest globally addressable program unit within mCAT. A task consists of at least one thread (separately executable program inside a task). You don't have to worry about threads if your tasks contain just one program.

2.2. Relation with other mCAT Concepts

As we heard before, a task includes one or more threads. It can call functions implemented in a shared library and send / receive messages to/from other tasks or interrupt drivers.

2.3. Task Related Data Structures

Every Task needs one global variable of type *INTEGER* called **Self.** Self is used to store the taskid, a 16-bit signed integer value. The *taskid* is needed to identify the task within various function calls. It is also needed in the C-Startup code (see file __*v4.c* in cc\lib).

2.3.1. The Data Structures before MCAT 2.20-R00168

> *Be aware that this data structure is documented for completeness and reference only! The structure should not be accessed directly! This structure is subject of change without notification in both size and structure!*

```
typedef struct {
      lword  type;        /* Type of accepted messages */
      lword  tree;        /* message queue */
      THREAD thread;      /* waiting thread */
} QUEUE;

typedef struct {
      IMD    imd;         /* IMD of this task */
      THREAD main;        /* pointer to main THREAD */
      lword  res0;        /* MUST BE NULL */
      lword  res1;        /* MUST BE NULL */
      lword  res2;        /* MUST BE NULL */
      lword  res3;        /* MUST BE NULL */
      word   flags;       /* Task feature / state flags */
      word   quemax;      /* number of available queue descriptors */
      word   quecnt;      /* number of used queue descriptors */
      QUEUE  queue[1];    /* array of queue descriptors */
} TASK;
```

2.3.2. The Data Structures for non TLCS platforms, MCAT 2.10-R00168 and later

> *Be aware that this data structure is documented for completeness and reference only! The structure should not be accessed directly! This structure is subject of change without notification in both size and structure!*

```
typedef struct {
    MCATMSG             *tree;      // the message tree
    UINT32              id;         // the ass. msgid
    THREAD              *waiting;   // the ass. thread
} MQUEUE;

typedef struct __mddesc__ {
    RTTI                rtti;
    RTTI                owner;
    union {
        struct __ws__     *ws;
        struct __task__   *tsk;
        struct _mthread   *thread;
        struct __mddesc__ *next;
        void              *owner;
    } link;
    UNSIGNED            max;        // maximum number of possible queues
    UNSIGNED            avail;      // number of allocated queues
    MQUEUE              queues[4];  // the queues, minimum 4
} MQUEUEDESC;

typedef struct __task__ {
      RTTI          rtti;         // runtime type identifier
      IMD           *imd;         // pointer to module identifier
      THREAD        *main;        // pointer to main thread
      UINT32        res0;         // place holder
      UINT32        res1;
      MQUEUEDESC     *qd;         // queues
} TASK;
```

2.4. Function Reference

A task has, like an interrupt driver, an *identification number* called "task id" or "task number". This id is assigned to the task during the TaskCreate kernel function call. A task is addressed by this *id* in contrast to a thread which cannot be addressed globally. For that reason it's possible to send a message to a task but not to a thread.

TaskCreate

Function: Creates a task for use by mCAT. At this time the object code must be resident in memory, it must contain a valid IMD (task header) and the program code for TaskInit and TaskMain. TaskCreate does not activate the task.

C-Prototype: **INTEGER TaskCreate (IMD *imd, INTEGER dir, INT16 *error, INTEGER queues);**

Arguments:

imd Pointer to a data structure *IMD* witch can be generated by the *CIMD.EXE* utility. See tools manual for more information.

dir Assign numbers form the bottom (0) or the top of the list (MAX-TASK, for ex. 15). Allows the two predefined values "FromBottom" or "FromTop" *FromBottom* is reserved for *BITBUS GBS TASK* only (as BITBUS GBS TASK must be task ‘0'), FromTop is for all other purposes.

Error code **SYS_ERR_OK** ok

SYS_ERR_OUT_OF_MEMORY No memory available

SYS_ERR_ID_OVERFLOW Can't allocate ID (no more tasks)

queues If greater than 0, your task will be able to support more than the default queue. A value of 0 assumes the default value of 1 queue, a one means one additional queue. You need more than the default queue if you plan to have specialized threads waiting for messages form a specific source.

Returns: Taskid, a 16 bit integer for task identification

Supported:

mCAT	All versions
Hardware	All

Comments: Within the mCAT runtime library there is a support function called TaskStartup. It has the same calling convention as TaskCreate. In contrast, TaskStartup automatically calls TaskActivate after successful creation of a Task.

TaskDelete

Function: Deletes a task (its main thread) and all of its threads. Stops operation and - in case a dynamic memory manager is installed - frees memory used for this task and its threads.

C-Prototype: **INTEGER TaskDelete (INTEGER taskid);**

Arguments: taskid Identification assigned to task by TaskCreate

Returns: Error code

Error code	**SYS_ERR_OK**	ok
	SYS_ERR_NO_TASK	Task does not exist

Supported:

mCAT	All versions
Hardware	All

Comments:

TaskActivate

Function: Activates a task, i.e. puts it into the list of tasks ready to receive the processor if their priority is the highest on the list. If the task being activated has a higher priority than the currently running task, the current task is interrupted and the newly activated task starts immediately. A task must have been created before it can be activated. The function TaskStartup both creates and activates a task.

TaskActivate activates the main thread of the task by calling *ThreadSignal*.

C-Prototype: **INTEGER TaskActivate (INTEGER taskid, UNSIGNED prio);**

Arguments:

taskid	Identification assigned to task by TaskCreate
prio	Priority to override the startup priority defined in the task header IMD.

Returns: Error code **SYS_ERR_NO_TASK** Task not existent

Supported:

mCAT	All versions
Hardware	All

Comments:

TaskSuspend

Function: All threads of this task are suspended, i.e. they are taken out of the ready list and are marked suspended. Signals or messages sent to suspended tasks are not queued. The only way to re-activate a suspended task is to use the TaskResume call. Use this call for debugging purposes.

TaskSuspend calls *ThreadSuspend* to resume all threads associated with the Task.

C-Prototype: **INTEGER TaskSuspend (INTEGER taskid);**

Arguments: taskid Identification assigned to task by TaskCreate

Returns: Error Code **SYS_ERR_NO_TASK** Task not existent

SYS_ERR_IS_SUSPENDED Task is already suspended.

Supported:

mCAT	All versions
Hardware	All

Comments:

TaskResume

Function: Puts all threads of this task into the ready list again. Used for tasks that had been suspended by TaskSuspend.

TaskResume calls *ThreadResume* to suspend all threads associated with the Task.

C-Prototype: **INTEGER TaskResume (INTEGER taskid);**

Arguments: taskid Identification assigned to task by TaskCreate

Returns: Error Code **SYS_ERR_NO_TASK** Task not existent

SYS_ERR_IS_NOT_SUSPENDED

Task is not suspended.

Supported:

mCAT	All versions
Hardware	All

Comments:

TaskGetState

Function: Returns information about the main thread. For more information see *ThreadGetState*.

C-Prototype: **INTEGER TaskGetState (INTEGER taskid);**

Arguments: taskid Identification assigned to task by TaskCreate

Returns: See ThreadGetState for more information

Supported:

mCAT	All versions
Hardware	All

Comments:

TaskGetPtr

Function:	Returns a pointer to a task descriptor. Advanced and system use only.	
C-Prototype:	**TASK *TaskGetPtr (INTEGER taskid);**	
Arguments:	taskid	Identification assigned to task by TaskCreate
Returns:	Pointer	Pointer to Task descriptor. If NULL, task is unknown.
Supported:	mCAT	All versions
	Hardware	All
Comments:		

3. mCAT Threads

3.1. The Concept of Threads

This section describes thread related functions. A thread is the smallest executable program unit in mCAT. It is not globally addressable and can only be used within tasks. A thread can be dedicated to a special queue (special message types) thus work on data that comes from a single source. It might be used to handle a certain peripheral or communication channel.

However, it is much more efficent to do the same job using a periodic event like a “TickerMsg” to do some event polling.

Threads can be in one of the following states:

1. SUSPENDED

It's not in the list of ready threads and marked as suspended. It can be activated only by a call to ThreadResume. Signals sent to a suspended thread are lost.

2. SLEEPING

A thread is sleeping if it was just created or if a ThreadSleep, ThreadSleepQueued or MsgWait call was issued. Contrary to a suspended thread, it will get back into the ready list by receiving a signal (a message).

3. DELAYED

A thread is waiting for a timeout to elapse inside a ThreadDelay call. Nothing but the elapsed timeout can remove the thread form this state.

4. READY

If a thread is not suspended and not in delayed state, it can change form sleeping state into the ready state when a signal (ThreadSignal, MsgSend et al.) is received. The thread is added to the ready list. The thread with the highest priority in the ready list is the currently running thread.

5. RUNNING

There can be only one running thread - the one on top priority in the ready list.

A thread can create child threads. Be careful when accessing variables form different threads. Use Protect/Unprotect for critical sections.

3.2. Relation with other mCAT Concepts

Threads are the basic program unit in mCAT. Every task contains one or more threads. The top most thread of a task is the *main thread*. A task without at least one thread is a dead task.

3.3. Thread Related Data Structures

3.3.1. MCAT 2.10 and later - TLCS900 Platform

Be aware that this data structure is documented for completeness and reference only! The structure should not be accessed directly! This structure is subject of change without notification in both size and structure!

```
typedef struct __thread {
      struct __thread      *next; /* used for internal management */
      struct __thread      *prev;
      lword  time;               /* used for timeout */
      lword  null;               /* reserved */
      byte   flags;              /* "most important flags" */
      byte   prio;               /* prio of thread */
      lword  taskid;             /* own task */
      struct __thread      *parent;     /* parent thread ptr. */
      struct __thread      *child;      /* child thread ptr. */
      lword  xsp;                /* act. stack pointer */
      lword  isp;                /* initial stack pointer */
      lword  bsp;                /* base of stack (lowest possible addr) */
      lword  ipc;                /* initial pc */
      byte   iprio;              /* initial priority */
      byte   res1;               /* reserved */
} THREAD;
```

3.3.2. mCAT 2.20 – non TLCS platform

```
typedef struct _mthread    {
    CPU_REGISTER           cState;      // CPU depending register model
      XT                   timer;       // timeout & round robin
    struct _mthread *next;       // scheduler list next
      struct _mthread      *prev;       // scheduler list prev
      struct _mthread      *parent;     // parent thread ptr.
      struct _mthread      *pnext;      // parent thread chain
      struct _mthread      *child;      // child thread ptr.
      UNSIGNED             sigq;        // signal queue
      UNSIGNED             flags;       // "most important flags"
      UNSIGNED             prio;        // prio of thread
      INTEGER              retval;      // retval of sleep
      INTEGER              taskid;      // own task
      INTEGER              qid;         // reply queue id
      HLIST                *hlist;      // argument list
      UINT32               bsp;         // base of stack (lowest possible addr)
      UINT32               isp;         // initial stack pointer
} THREAD;
```

3.4. Function Reference

ThreadCreate

Function: Creates a thread for use by mCAT. The function will generate a descriptor structure (THREAD) and put the thread to the sleeping state: it must receive a signal to run. ***The new thread is created in the task space of the current task.***

C-Prototype: **THREAD *ThreadCreate (INTEGER (*start)(), long stacksize, UNSIGNED priority);**

Arguments:

start	Pointer to start of thread code (label start)
stacksize	Required size of stack in bytes. Suggested value: 1024
priority	Starting priority of thread

Returns:

Pointer	Pointer to the thread descriptor or NULL if call fails

Supported:

mCAT	mCAT2 and higher
Hardware	All

Comments:

ThreadCreateEx

Function: Creates a thread for use by mCAT. The function will generate a descriptor structure (THREAD) and put the thread to the sleeping state: it must receive a signal to run. ***The new thread is created in the task space of the current task.*** The difference between *ThreadCreate* and *ThreadCreateEx* is that you can pass an optional argument with *ThreadCreateEx*. This argument is passed to the created thread function as an argument.

> *Please note that the thread function MUST be declared using the SYS_FDECL macro! If you don't use it, you may not receive the argument of the TLCS platform:*
>
> *INTEGER SYS_FDECL threadfunction(void *arg)*

C-Prototype: **THREAD *ThreadCreateEx (INTEGER (*start)(), long stacksize, UNSIGNED priority, void* args);**

Arguments:

start	Pointer to start of thread code (label start)
stacksize	Required size of stack in bytes. Suggested value: 1024
priority	Starting priority of thread
args	Pointer passed to the thread function as an argument

Returns: Pointer Pointer to the thread descriptor or NULL if call fails

Supported:

mCAT	mCAT2.10-R00168 and higher
Hardware	All

Comments:

ThreadKill

Function: Deletes a thread and all of its child threads. Stops operation and frees memory used for stack and descriptor.

C-Prototype: **void ThreadKill (THREAD *thread);**

Arguments: thread Pointer to thread descriptor. Assigned to thread by Thread-Create

Returns: none

Supported:

mCAT	All versions
Hardware	All

Comments:

ThreadSuspend

Function: Suspends a thread. The thread will ignore all signals until it is resumed.

C-Prototype: **void ThreadSuspend (THREAD *thread);**

Arguments: thread Pointer to thread descriptor. Assigned to thread by Thread-Create

Returns: none

Supported:

mCAT	All versions
Hardware	All

Comments:

ThreadResume

Function: Will remove the suspended state. The thread will be able to receive signals again and if it was ready or running before it was put into suspend state, it will be inserted into the ready list again, too. All signals and possible timeouts are lost.

C-Prototype: **void ThreadResume (THREAD *thread);**

Arguments: thread Pointer to thread descriptor. Assigned to thread by Thread-Create

Returns: none

Supported:

mCAT	All versions
Hardware	All

Comments:

ThreadSetPrio

Function: A new priority is assigned to the thread. If it is in the list of ready threads, it will be removed and re-inserted at the new priority. If the thread is suspended or sleeping, only the priority will be changed.

C-Prototype: **void ThreadSetPrio (THREAD *thread, UNSIGNED prio, INTEGER fixed);**

Arguments:

thread	Pointer to thread descriptor. Assigned to thread by Thread-Create
prio	New priority.
fixed	If fixed is true, the thread is set in fixed priority mode at priority *prio*. The Priority can not be boosted by a ThreadSignal or by receiving a higher priority message.

Returns: none

Supported:

mCAT	All versions
Hardware	All

Comments:

ThreadGetPrio

Function: Returns the current priority of the thread.

C-Prototype: **UNSIGNED ThreadGetPrio(THREAD *thread);**

Arguments:

thread	Pointer to thread descriptor. Assigned to thread by Thread-Create

Returns: The current priority of the *thread.*

Supported:

mCAT	All versions
Hardware	All

Comments:

ThreadGetState

Function:	Returns information about the thread.	
C-Prototype:	**INTEGER ThreadGetState (THREAD *thread);**	
Arguments:	thread	Pointer to thread descriptor. Assigned to thread by Thread-Create
Returns:	retval	retval & **THREAD_SUSPENDED** retval & **THREAD_DELAYED** retval & **THREAD_FIXED_PRIO** retval & **THREAD_IN_TQUE** retval & **THREAD_READY**

Supported:

mCAT	All versions
Hardware	All

Comments:

THREAD_SUSPENDED	The Thread is suspended
THREAD_DELAYED	The Thread is waiting inside a ThreadDelay call for the timeout to elapse
THREAD_FIXED_PRIO	The Thread is running at a fixed priority
THREAD_IN_TQUE	The Thread is sleeping (THREAD_READY is NOT set) and timeout is not 0. The Thread is in the timeout queue.
THREAD_ READY	Thread is in the Thread list, ready to run. This is the READY STATE

ThreadSlice

Function: If there's at least one other thread in the ready list whose priority is equal to that of the running thread, the next one will become ready instead.

C-Prototype: **void ThreadSlice (void);**

Arguments: None

Returns: None

Supported:

mCAT	All versions up to 2.08. Not available in later releases.
Hardware	All

Comments:

> *This call should not be used anymore. It is included here for completness only. This call will not be supported in mCAT 2.1 and higher.*

ThreadSleep

Function: The running thread is put to the sleeping state and the next one in the ready list gets the processor. The now sleeping thread will be waken up by a signal send to it. If ms is not zero, the sleeping thread will be signalled after ms milliseconds. The function is used by mCAT *message passing* (MsgWait).

C-Prototype: **INTEGER ThreadSleep (UINT32 ms);**

Arguments: ms Milliseconds to being signalled. Zero for infinite sleep. This parameter is not supported in mCAT Versions lower than 2.06. With mCAT 2.10-R00168 use the macro **SYS_WAIT_INFINITE** to signal an infinite wait instead of “0”.

Returns: Error Code

SYS_ERR_OK	The thread was successfully signaled
SYS_ERR_TIME_OUT	The timeout elapsed befor a signal was received

Supported:

mCAT	Timeout supported with 2.06 and later Error codes supported with 2.07 and later
Hardware	All

Comments: Please check the alternate function *ThreadSleepQueued* for more options!

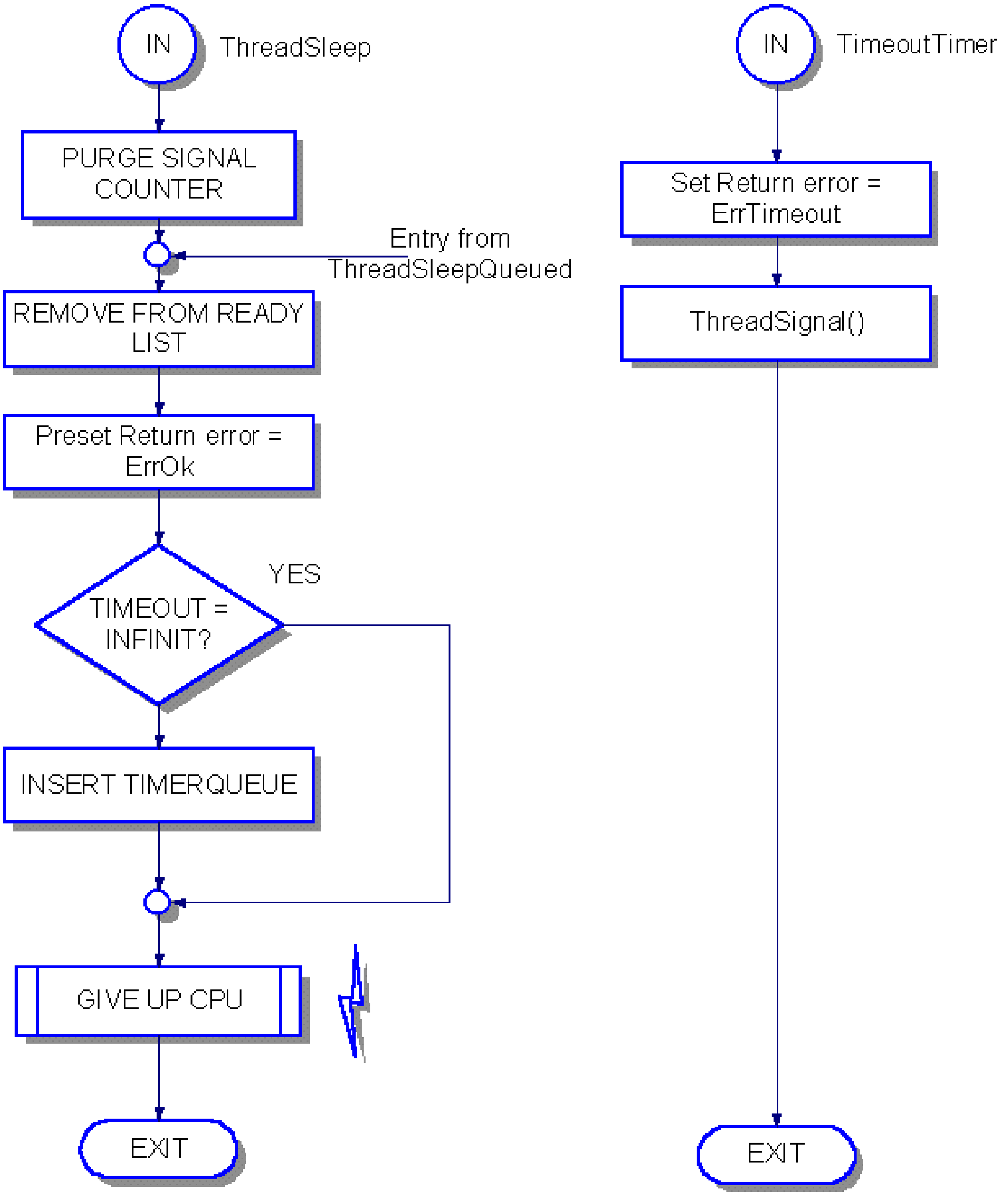


Figure 10: ThreadSleep

Figure 1 Shows the control flow inside a ThreadSleep call. First, the signal counter is cleared and the return code is preset to **SYS_ERR_OK**. Then, if a timeout is requested, the thread is placed into the timeout queue (a background timer keeps control over the timeout queue). Then the control over the CPU is given up. If a signal is received (the thread maybe signaled by the timeout timer process or by another thread), the thread will get in control at exactly the point where it gave it up earlier.

ThreadSleepQueued

Function:

This function is a derivative of “ThreadSleep”. If a function is signaled by use of ThreadSignal before the signaled thread entered ThreadSleep(), the signal is lost.

With ThreadSleepQueued, the incoming signals are counted (“queued”) and you have now two more alternatives:

1. ThreadSleepQueued (tout, TRUE)

The signal(s) are no longer lost. When you enter ThreadSleepQueued and there is already a signal pending, your current thread will return immediately – it will NOT sleep! The signal queue is flushed (the counter set to 0) and other signals lost.

2. ThreadSleepQueued (tout, FALSE)

Again: When you enter ThreadSleepQueued and there is already a signal pending, the function will be terminated immediately and returns to the calling program. However, the signal queue is **not** flushed. The signal is dequeued (counter is decremented).

C-Prototype: **INTEGER ThreadSleepQueued (UINT32 ms, INTEGER flush);**

Arguments:

ms	Milliseconds to being signalled. Zero for infinite sleep. This parameter is not supported in mCAT versions lower than 2.06. With mCAT 2.10-R00168 use the macro **SYS_WAIT_INFINITE** to signal an infinite wait instead of “0”.	
flush	**TRUE**:	flush signal queue before leaving ThreadSleepQueued
	FALSE:	do NOT flush signal queue before leaving ThreadSleepQueued

Returns:

Error Code	**SYS_ERR_OK**	The thread was successfully signalled
	SYS_ERR_TIME_OUT	The timeout elapsed before a signal was received

Supported:

mCAT	mCAT 2.09 and later
Hardware	All

Comments:

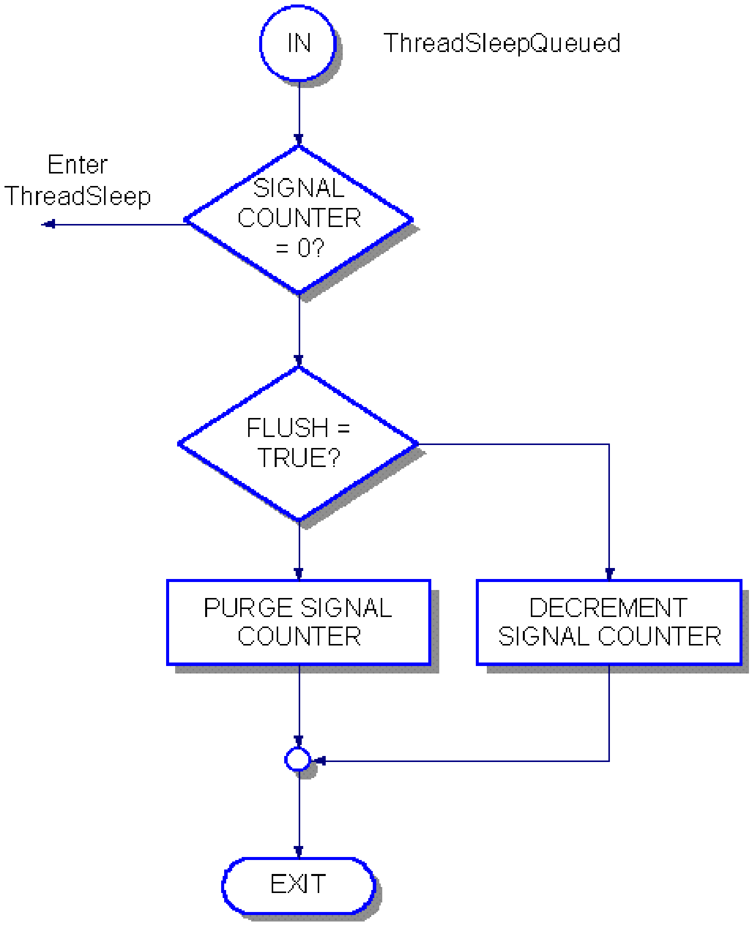


Figure 11: ThreadSleepQueued

Figure 2 Shows the control flow of a ThreadSleepQueued call. If no signal is pending, it just branches to enter the std. ThreadSleep call. The difference to ThreadSleep is only important if there are pending signals! Depending on the value of argument flush, the signal counter is cleared or just decremented.

Anyway, the control returns to the calling thread immediately.

ThreadDelay

Function:	ThreadDelay works the same way ThreadSleep does but with one little difference: While the process is in a DELAY state, it will accept no signals (by ThreadSignal or MsgSend) except the internal timeout expired. This allows a thread to sleep for a defined delay without unexpected interruption.	
C-Prototype:	**void ThreadDelay (UINT32 ms);**	
Arguments:	ms	Milliseconds to being signalled. Zero for infinite sleep. Please note that your thread will NEVER wakeup again if you call this function with ms = **SYS_WAIT_INFINITE**.
Returns:	None	

Supported:

mCAT	All versions
Hardware	All

Comments:

ThreadSignal

Function:	Puts a thread into the ready list (after ThreadCreate or ThreadSleep) using the priority given as an argument or with no change in priority if this parameter is zero. If the thread is already ready and the current priority of the thread is lower than the new priority, the threads priority is boosted to the new value.	
C-Prototype:	**void ThreadSignal (THREAD *thread, UNSIGNED prio);**	
Arguments:	thread	Pointer to thread descriptor. Assigned to thread by ThreadCreate
	prio	New priority or zero for no change.
Returns:	none	

Supported:

mCAT	All versions
Hardware	All

Comments:

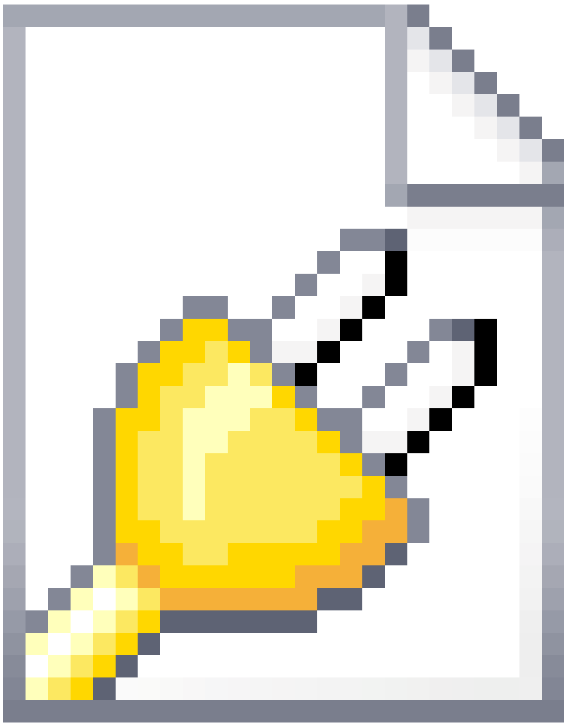


Figure 12: ThreadSignal

Legend:

IN: entry point of a ThreadSignal call.

fixed prio: Is the thread in fixed priority mode? This mode can be set using TreadSetPriority() only.

priority: The current priority of the thread.

READY: A thread is READY if it is in the list of ready threads.

INSERT: Insert a thread into the READY list

REMOVE: Remove a thread from READY list

TopOfList: The thread with the highest priority is always “TopOfList”.

SWITCH: Perform a thread-switch: Make new thread the running thread.

ThreadProtect

Function: This function protects multitasking/multithreading. This call is used before entering a critical section of code that must not be interrupted by other Tasks / Threads.

C-Prototype: **void ThreadProtect (void);**

Arguments: None

Returns: None

Supported:

mCAT	All versions
Hardware	All

Comments: For compatibility with mCAT 1.xx the macros Protect and UnProtect are provided.

```
#define Protect()       ThreadProtect()
```

ThreadUnProtect

Function: Ends the critical section entered with ThreadProtect. With versions prio to 2.09, *ThreadUnProtect()* unprotects multitasking/multithreading unconditionally. With versions below 2.09, UnProtect re-enables multithreading unconditionally. With 2.09, an internal counter is maintained to be sure that UnProtect re-enables multithreading only if it was called as often as Protect. This was introduced to allow nested calls to Protect/UnProtect!

C-Prototype: **void ThreadUnProtect (void);**

Arguments: None

Returns: None

Supported:

mCAT	All versions
Hardware	All

Comments: For compatibility with mCAT 1.xx the macros Protect and UnProtect are provided.

```
#define UnProtect()   ThreadUnProtect()
```

ThreadCreateKrnl

Function: Creates a thread for use by mCAT. The function will generate a descriptor structure (THREAD) and put the thread to the sleeping state: it must receive a signal to run. ***The new thread is created in the task space of the kernel task instead of the task space of the current task.*** This call can be used to create threads without a task. Usually this call is used to create background threads for use inside of libraries or interrupt drivers.

C-Prototype: **THREAD *ThreadCreateKrnl (INTEGER (*start)(), long stacksize, UNSIGNED priority, void* args);**

Arguments:

start	Pointer to start of thread code (label start)
stacksize	Required size of stack in bytes. Suggested value: 200
priority	Starting priority of thread
args	Pointer passed to the thread function as an argument

Returns: Pointer Pointer to the thread descriptor or NULL if call fails

Supported:

mCAT	mCAT2.09 and higher
Hardware	All

Comments:

Example:

```
typedef struct {
      UINT32 delay;
      UINT8        *out;
} ThreadArgs;

INTEGER MyThread ( ThreadArgs *args )
{
      while(args) {
             ThreadDelay(args->delay);  // sleep configured time
             *out = ~(*out);                    // invert value .. just to do some-
thing
      }
}

...
      ThreadArgs args;
      THREAD *thread;
      args.delay = 100; args.out = (UINT8 *) 0xffffff;      // some arguments
      thread = ThreadCreateKrnl(MyThread,400,128,&args);
      if (thread) {
             ThreadSignal(thread,128);
             ....
```

ThreadSetHandler

Function: The **ThreadSetHandler** function is passed the address of a function (*func*) to be called when the program terminates. Successive calls to **ThreadSetHandler** create a register of functions that are executed in LIFO (last-in-first-out) order. The functions passed to **ThreadSetHandler** take the argument ***args*** as a parameters. **ThreadSetHandler** uses the heap to hold the register of functions. Thus, the number of functions that can be registered is limited only by heap memory.

C-Prototype: **void *ThreadSetHandler (INTEGER type, void (*func)(), void* args);**

Arguments:

type	Selects the type of handler. Currently only exit handlers are supported. Therefore set this argument to *KRNL_HDL_TYPE_ATEXIT*
func	Function to be called
args	Optional argument for func()

Returns: Pointer — Pointer to an internal data structure on success or NULL if call fails

Supported:

mCAT	MCAT2.10 R00013 and higher
Hardware	All

Comments: **This function is used to implement std. ANSI-C function *atexit().***

```
void __cdecl exit_func (char *args)
{
      WrStr(“atexit: “);
      WrStr(args);
      WrLn();
}

void TaskMain()
{
      ThreadSetHandler(KRNL_HDL_TYPE_ATEXIT, exit_func, “task deleted”);
      TaskDelete(Self);
}

Output:

2+>init 402000
atexit:  task deleted
2+>
```

4. mCAT Message Passing

4.1. The Concept of Message Passing

A message is a data structure that is declared and filled by the sender. It is send to the receivers message queue. There is no real data transfer happening when a message is send, instead, the receiving task is send a signal (see ThreadSignal) with a priority attached to it.

If the receiving task is waiting for a message, it is signaled and its priority is set to the priority of the message received. If the receiver is not sleeping, its priority is boosted to the priority of the message to prevent an effect know as ***priority inversion***.

Pointers to the message are put into the receivers queue which actually is just a list of message pointers sorted by priority.

To differentiate between messages of various origins, a message type code (msg.type) is provided by the server task. A client can query the type code to request a defined functionality.

mCAT 2 supports multiple message queues. A separate queue can be created for a specific type code only. All messages that do not have a dedicated queue are queued in the default queue (queueid = 0). Separate queues can make life easier sometimes.

4.2. Relation with other mCAT Concepts

The functions of the MSG (message) group are closely related to the TASK group. They build the fundamental operating mechanism for mCAT.

4.3. The MSGID & MSG Data Structures

Be aware that this data structure is documented for completeness and reference only! It should not be accessed directly! This structure is subject of change without notification in both size and structure!

```
typedef struct {
      UINT32       next;         /* internal link */
      UINT32       link;         /* reserved */
      UINT32       name;         /* pointer to name */
      INTEGER      (*cnvt)();    /* reserved for future use */
      UNSIGNED     node;         /* node information, currently not used
                                    With mCAT 210T00106 0 is defined and
                                        set for local MSGID's */
      INTEGER      hdl;          /* task / interrupt ident of server */
      UINT32       id;           /* type stored in the message header */
} MSGID;
```

An application message can be formed using the mCAT message header “MSG”.

```
typedef struct {
      UINT32       right;        /* internal link */
      UINT32       left;         /* internal link */
      UINT16       len;          /* length over all */
      UINT32       type;         /* “id” from MSGID structure */
      UINT16       net;          /* internal routing information */
      INT16        src;          /* requesters task / interrupt ident */
      UINT8        prio;         /* priority requester->server */
      UINT8        reply;        /* priority server->requester */
      UINT16       error;        /* error code */
} MSG;
```

A simple example:

```
typedef struct {
      MSG    hdr;                /* the std. mCAT message header */
      INT32  part_counter;       /* some application data */
} MyMSG;
```

You can now send and receive messages of type MyMSG. Where ever “MSG *” is used as an argument to a kernel function, you can now use your own message:

```
MyMSG  report;
MsgSendRequest(&report.hdr,Self,ReportId,200,200);
```

4.4. Messages: How to?

1. Create a C-Header file (“mymsges.h”)

   - Design and include your message structure(s).

```
typedef struct {
       MSG    hdr;                 /* the std. mCAT message header */
       UINT32 part_counter;        /* some application data */
} MyMSG;
```

   - name your message

```
#define MyMSGID     “my/part_counter”
```

2. Write the server

   - Include “mymsges.h”

   - Use MsgIdCreate() to register the MSGID

```
my_msg_id = MsgIdCreate (Self,  MyMSGID, NULL);
```

   - Write the code to handle the message

3. Write the client code

   - Include “mymsges.h”

   - Use MsgIdQuery() to query for the server

```
my_msg_id = MsgIdQuery (MyMSGID);
```

   - Write the code to send requests to the server

4.5. Flow Charts

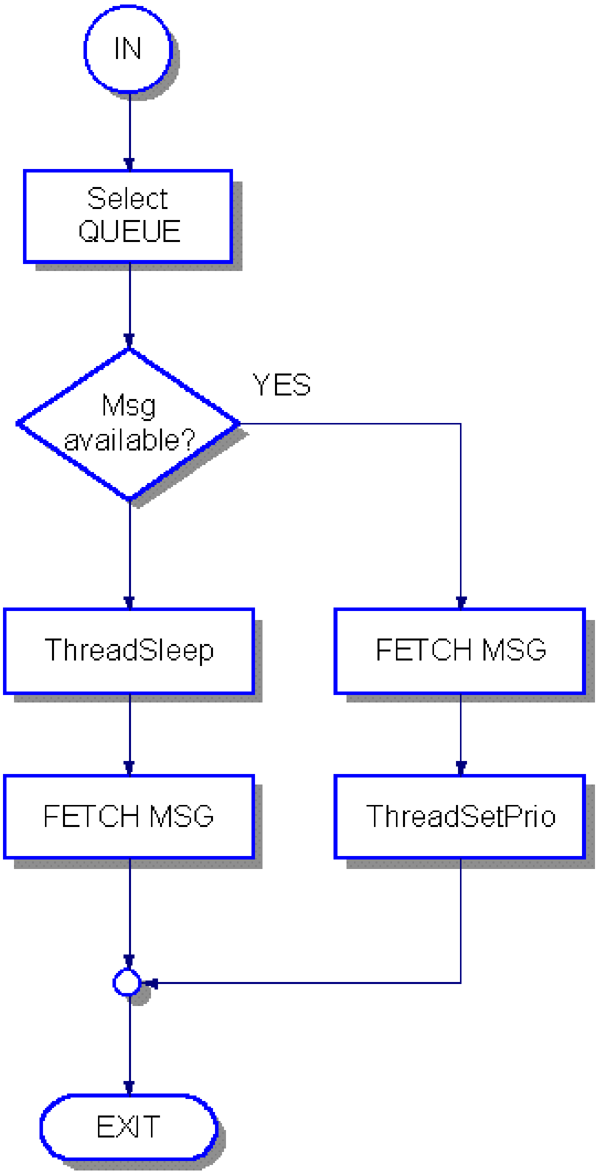


Figure 13: MsgWait

Figure 4 Shows the control flow inside a MsgWait call. After selecting the requested queue MsgWait checks if there is at least one message pending.

If so, it fetches the message from the queue and sets the current thread's priority to the message priority and returns.

If not, the current thread is send to sleep. It will be signaled again (wake up) if a timeout occurs, the thread is explicitly signaled from another thread or implicitly signaled by a MsgSend type call (see Figure 5, page 26). No matter why the Thread was signaled, MsgWait then tries to fetch the most recent message from the queue. If there is a message, MsgWait returns the pointer to this message. If not it returns NULL.

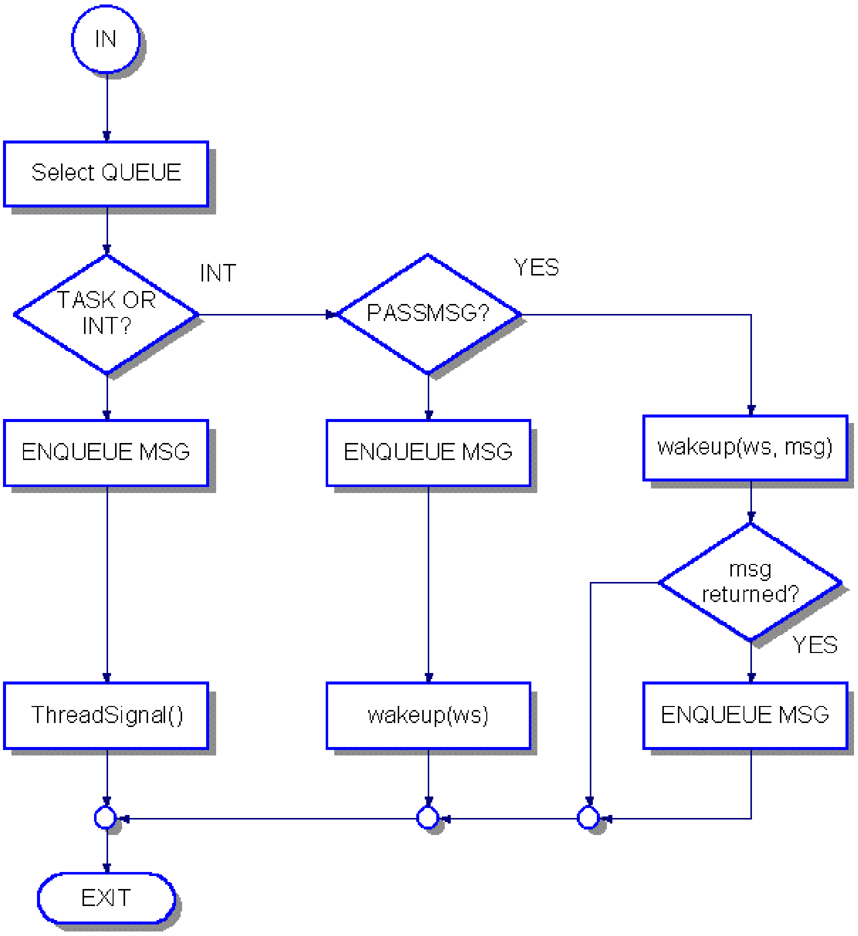


Figure 14: MsgSend

Figure 5 Shows the control flow of a MsgSend type call (MsgSend, MsgSendReply, or MsgSendRequest). After selecting the destination queue, the flow is different for tasks and interrupt drivers.

If the destination is a task, the message is inserted into the tasks message queue and the associated thread is signalled. This is what we called an implicit signal. This is the signal that wakes up a sleeping thread inside a MsgWait call (See Figure 4, page 25).

If the destination is an interrupt driver and the version of mCAT is below 2.08, the message is inserted into the interrupt drivers message queue and the interrupt drivers *wakeup* function is called (if present) to inform the driver about the new message (PASSMSG=NO).

For mCAT 2.08 or later, it is also possible to use a option called PASSMSG. In this case *wakeup* is called first with the message as an argument. If *wakeup* returns the message pointer, the message will be inserted into the queue, if it returns NULL it will not. See chapter 5 for more information.

Because PASSMSG=YES has some advantages, it is the preferred methodology. On other platforms than the TLCS platform, PASSMSG=YES is the only option available!

4.6. Function Reference

MsgIdCreate

Function: To register a new message type, mCAT has to be informed about the type name and will in turn assign a handle or type id that is called msgid. The name of the message can contain any standard ASCII character and there is no length restriction (however, it is recommended not to use strings longer than 32 byte). Please don't start your names with "mCAT/" as system messages use this prefix. It is a good practice to use a per-personal prefix for your applications (“my-prefix/my-app/do-this-and-that”).

C-Prototype: **MSGID *MsgIdCreate (INTEGER self, char *name, MSGID *msgid);**

Arguments:

self	ID of creating task
name	User defined name
msgid	If NULL, MSGID is allocated automatically If NOT NULL, "msgid" is used to store information. First use is preferred. This argument will not be supported in future versions (> 2.09).

Returns: Pointer — Pointer to message id structure

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgIdQuery

Function: Use MsgIdQuery to find out the message id (msgid) of an already defined message type by name. Usually the client task will need to know the msgid of the server-defined message type.

C-Prototype: **MSGID MsgIdQuery (char *name)**

Arguments: name — Name to be searched for.

Returns: Pointer — Pointer to message id, NULL if name is not found.

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgSendRequest

Function: A message is sent as a request to a task or interrupt handler, awaiting a reply. The message header is filled automatically.

C-Prototype: **INTEGER MsgSendRequest (MSG *msg, INTEGER self, MSGID *msgid, UNSIGNED prio, UNSIGNED reply);**

Arguments:

msg	Pointer to message to be sent
self	Task id of sending task (self)
msgid	Pointer to message id structure
prio	Priority assigned to this message. Value between 1 (low) and 255.
reply	Priority requested for reply. The receiver usually moves this value to the priority field of the reply message (automatically when using MsgSendReply). Special values for "reply": 0: This message IS a reply! Must not be used in a MsgSendRequest call! FFh: Should not be used.

Returns: Error Code

SYS_ERR_OK	Ok
SYS_ERR_NO_TASK	Destination task not existent
SYS_ERR_NO_TARGET	Destination interrupt handler not existent

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgSendReply

Function: A msg is replied to the requesting task. The major duty of MsgSendReply is to copy the field msg->reply of the message header to msg->prio and then to clear msg->reply. Then the msg is sent to msg->src. The own taskid "Self" is used to prevent an infinite handling of a message if msg->src is equal to Self!

C-Prototype: **INTEGER MsgSendReply (MSG *msg, INTEGER self, UNSIGNED error);**

Arguments:

msg	Pointer to reply message
self	Own taskid, necessary to prevent deadlocks
error	Reply code sent to requester:

ACK	Reply ok
NAK	Error (cannot interpret message).

Will be filled into msg->error. Beside of ACK and NAK, a user defined error code can be sent. With mCAT 2.00 the error field is 16-Bit.

Returns: Error Code

SYS_ERR_OK	Ok
SYS_ERR_NO_TASK	Destination task not existent
SYS_ERR_NO_TARGET	Destination interrupt handler not existent

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgSend

Function: Send a message to task taskid. The user is responsible for filling the message header.

C-Prototype: **INTEGER MsgSend (INTEGER taskid, MSG *msg);**

Arguments:

taskid	ID code of destination task.
msg	Pointer to the message to be sent.

Returns:

Error Code	**SYS_ERR_OK**	Ok
	SYS_ERR_NO_TASK	Destination task not existent
	SYS_ERR_NO_TARGET	Destination interrupt handler not existent

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgPost

Function: Send a message to task taskid just as with MsgSend however without issuing a signal to the destination task. This prevents task switching and allows the source task to continue even if the message sent has a higher priority than itself.

C-Prototype: **INTEGER MsgPost (INTEGER taskid, MSG *msg);**

Arguments:

taskid	ID code of destination task.
msg	Pointer to the message to be posted.

Returns:

SYS_ERR_OK	Ok
SYS_ERR_NO_TASK	Destination task not existent
SYS_ERR_NO_TARGET	Destination interrupt handler not existent

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgWait

Function: Put yourself to the sleeping state until a message arrives at queue *hdl* or the timeout *timeout* has passed. A call to MsgWait effectively stops task execution and lets other tasks run until the desired message is received. If a message with sufficient priority is waiting at the queue, task execution continues. A timeout condition can be checked by testing the message pointer to be equal to NULL.

C-Prototype: **MSG *MsgWait (INTEGER hdl, UINT32 timeout);**

Arguments:

hdl	Handle associated with the queue this call waits for. Zero if this call waits for the default queue. A specific handle (assigned by MsgAddQueue) if we wait at a queue specialized on one specific message type.
timeout	Timeout length in milliseconds. INFINITE if no timeout. This parameter is not supported in mCAT Versions lower than 2.06. With mCAT 2.10-R00168 and higher use the macro **SYS_WAIT_INFINITE** to signal an infinite wait instead of “0” in previous versions.

Returns:

Pointer	Pointer to incoming message OR NULL if the desired queue doesn't exist OR if timeout elapsed.

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgUpdate

Function: A message is sent as a request to a task or interrupt handler, awaiting a reply. The reply will NOT be send to the tasks queues as usual but send to a hidden queue private to the current thread. After sending the request, MsgUpdate waits for the reply in front of the hidden queue.

This function provides a totally easy and save method to send and receive a message in a client application with a single function call.

The message header is filled automatically. The priority of the calling thread is not changed, the request is send with the current threads priority.

You should be very careful using a timeout with this function. A timeout will signal a fatal system failure and should be handled like this!

C-Prototype: **MSG *MsgUpdate (MSG *msg, MSGID *msgid, UINT32 timeout);**

Arguments:

msg	Pointer to message to be sent
msgid	Pointer to message id structure
timeout	Timeout length in milliseconds. **SYS_WAIT_INFINITE** to wait infinite.

Returns:

MSG *	Pointer to the replied message or NULL if TIMEOUT or fail

Supported:

mCAT	2.10 and higher
Hardware	All

Comments:

This function can not be used for BITBUS master applications – the BITBUS driver can not handle the hidden queues yet.

Be sure not to send a message to your own task when your current thread is intended to handle those types of messages! A DEADLOCK will occur!

Must not be used inside interrupt handlers!

MsgGet

Function: Read a message from queue *hdl*. If “hdl” is zero, we read from the default queue. This is a MsgWait without timeout and total freedom to access any task or interrupt driver (because taskid is given). This call is needed inside an interrupt driver to fetch a message from its queues.

C-Prototype: **MSG *MsgGet (INTEGER taskid, INTEGER hdl);**

Arguments:

taskid	Own taskid
hdl	Handle associated with the queue this call queries. Zero if this call goes to the default queue. A specific handle (assigned by MsgAddQueue) if we read from a queue specialized on one specific message type.

Returns:

Pointer	Pointer to desired message or NULL if no message available.

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgAddQueue

Function: If you have created your tasks with the queues parameter greater than 1, multiple queues are allowed. To use them, the corresponding message types (MSGID) to be handled by these queues must be defined. This is the task of MsgAddQueue. It returns a queue handle associated with a specific message type. This handle is used in the MsgWait and MsgGet functions.

C-Prototype: **INTEGER MsgAddQueue (INTEGER taskid, UINT32 msgid);**

Arguments:

taskid	Own taskid
msgid	Message type MSGID to associate with the queue. Msgid is the type code stored in msg.type / msgid.id. To get it, use MsgIdQuery.

Returns: Handle or zero for error (no more queues available, check TaskCreate).

Supported:

mCAT	All versions
Hardware	All

Comments:

MsgDelQueue

Function: Free the queue specified by *hdl*. All waiting requests are sent back to their owners (error = **NAK**), replies are ignored. A waiting thread is signalled at the current priority level. MsgWait will return 0 (not supported). This function is only used for debugging.

C-Prototype: **INTEGER MsgDelQueue (INTEGER taskid, INTEGER hdl);**

Arguments:

taskid	Own taskid
hdl	Queue handle

Returns: Error Code **SYS_ERR_OK**

Supported:

mCAT	All versions
Hardware	All

Comments:

5. mCAT Interrupts and QuickISR Interrupts

5.1. The Concept of Interrupt Drivers

To understand an mCAT interrupt driver you have to learn about three functions and one data structure. The functions are:

1. “service”: The interrupt service routine
2. “wakeup”: The message service routine
3. “go”: The driver enable call

5.1.1. The WorkSpace Data Structure

The interrupt driver workspace structure is needed to store data needed to manage the interrupt driver. For example a pointer to the private interrupt stack is included as well as the queues to store messages send to the interrupt driver. The interrupts workspace is comparable to a tasks TASK structure.

All service routines (service, wakeup and go) are called with the WS pointer as argument. This may help to distinguish between several interrupt sources handled by the same code as well as it helps to write drivers that do not need “static” variables in RAM. How does this work?

If you add your variables to the end of a WS structure like this

```
typedef struct {
      WS     ws;
      long   counter;
} MyWS;
```

you can allocate the entire structure by just calling IntInstall() with the size of your structure (sizeof(MyWS)) instead of calling it with (sizeof(WS)). Your variables are now in the dynamically allocated RAM area and you must no longer worry where to place them. *All mCAT drivers are written in this way*.

5.1.2. WS Data Structure on the TLCS platform

Be aware that this data structure is documented for completeness and reference only! The structure should not be accessed directly! This structure is subject of change without notification!

HOWEVER, THE SIZE OF THIS STRUCTURE WAS FIXED TO 4EH WITH ALL mCAT UP TO 2.09!

```
typedef struct {
      IMD           imd;                /* IMD */
      int           (*service)();       /* ASM-service call */
      int           (*notify)();        /* notify function */
      int           (*cservice)();      /* c-service call */
      int           (*wakeup)();        /* wakeup function */
      int           (*go)();            /* called after system boot */
      word          flags;              /* flags field */
      word          quemax;             /* number of queues (3) */
      word          quecnt;             /* number of used queues */
      QUEUE         queue[3];           /* a interrupt driver can have 3 queues */
      void          *stack;             /* interrupt stack pointer */
      word          intque;             /* reserved for future use */
      word          wres;               /* reserved for future use */
      long          lres;               /* reserved for future use */
} WS;
```

5.1.3. WS Data Structure on non TLCS platforms

Be aware that this data structure is documented for completeness and reference only! The structure should not be accessed directly! This structure is subject of change without notification!

```
typedef struct __ws__ {
      RTTI          rtti;               // runtime type identifier
      IMD           *imd;               // pointer to module identifier
      INTEGER        (*service)();      // pointer to service routine
      MSG *          (*wakeup)();       // wakeup pointer
      INTEGER        (*go)();           // enable pointer
      MQUEUEDESC     *qd;               // queues
} WS;
```

5.1.4. The “service” Function

service

Function: Interrupt service routine (ISR). Is called after an interrupt occurred. The stack is the local interrupt stack created inside the IntInstall() call, interrupts of same and lower levels are disabled. The pointer to the interrupts workspace is passed. This function can be declared as “__adecl” with C-Compiler version 4 or higher.

You may extend WS with your application specific data. Example:

```
typedef struct {
       WS           ws;
       long         product_count;
} MyWS;

void SYS_FDECL service (MyWS *ws)
{
}
```

You may call MsgGet() to retrieve a message or any “MsgSend” type function to send a message to another interrupt driver or task.

C-Prototype: **void SYS_FDECL service (WS *ws);**

Arguments: ws Pointer to the allocated workspace

Returns: None

Supported:

mCAT	All versions
Hardware	All

Comments:

5.1.5. The “wakeup” Function

For an interrupt driver at least “service” has to be implemented. A *wakeup* function is needed to inform the driver if a message was received. This must be explained a bit more precisely. First, let us classify interrupt drivers into the following basic classes:

1. periodic
2. input
3. output

It depends on the class of driver whether you need a wakeup function or not:

A *periodic* class interrupt driver receives an interrupt from the hardware once every time slice. No software interaction is needed to make sure the next interrupt will be issued. Example: A periodic timer interrupt. You need no wakeup function.

An *input* class interrupt driver receives an interrupt from the hardware when ever a data item gets available. No software interaction is needed to make sure the next interrupt will be issued after data is read. Example: A UART receiving characters. You need no wakeup function (but sometimes you may use it anyway, because it can make a complex design simpler)!

An output class interrupt driver sends data to a hardware device and receives an interrupt in turn to output the next unit of data. Example: Sending a string to a serial port. When all data is send, the final interrupt can not be served anymore, it is lost. Sometimes we speak of a *sleeping state* the driver is in. No other interrupt will occur until not another data unit is output. But who shall do this? Right, it is the *Wakeup()* call. New data arrives and needs to be handled. In our example, *Wakeup()* would output the first character of the new string. After sending the first character a new interrupt would occur and the normal output mechanism would handle the rest of the string. *Wakeup()* may look like this:

```
MSG *Wakeup(MYWS *ws, MSG *in)
{
        if (ws->tx.sleeping) {
                // ISR sleep state, start transmission
                output_one_byte(ws,in->data[0]);
                // save message as current under service
                ws->tx.current_output_buffer = in;      // message top be served
                ws->tx.current_output_counter = 1;      // first char already sent
                // exit sleeping state
                ws->tx.sleeping = FALSE;
                in = NULL;    // in is under service, return NULL!
        } /* endif */

        return in;
}
```

With mCAT 2.08 we introduced a new Option: PASSMSG mode for the wakeup function. If enabled, the message is not enqueued but it is passed to the wakeup function as a second argument. If it can be handled by the wakeup function, wakeup returns NULL. If not, wakeup returns the message – it will be queued in this case. This mode make things easier in some cases. For the different control flows of *wakeup* function with and without PASSMODE option, please refer to Figure 5: Loading a SHX-File from the ComamndLline, page 26.

wakeup

Function: Do the first data output operation to start the interrupt procedure. Be aware that interrupts are disabled up to and including level 4 while calling wakeup. Do not re-enable interrupts and keep wakeup as short and fast as possible.

C-Prototype: **MSG * SYS_FDECL wakeup (WS *ws, MSG *in); (PASSMSG mode)**

void wakeup (WS *ws); (std. mode)

Arguments:

ws	The workspace pointer
in	Pointer to the message just received. PASSMSG mode only. If the message can be handled immediately inside wakeup, return NULL. if it can not be handled return the message – it will be queued then! This will reduce the need to call “MsgGet” in some cases and speed up things a bit. However, be careful not to handle messages in wakeup while others still waiting in the queue (MsgGet).

Returns: None

Supported:

mCAT	All versions. PASSMSG mode is supported in mCAT 2.08 or higher.
Hardware	All

Comments:

5.1.6. The "go" Function

go

Function:	Called after system initialisation, just before the system enables interrupts. Used to finally enable hardware.	
C-Prototype:	**void go (WS *ws);**	
Arguments:	ws	The workspace pointer
Returns:	None	
Supported:	mCAT	All versions.
	Hardware	All
Comments:		

5.1.7. The "notify" Function

The notify function has been supported in all mCAT Versions up to and including 2.09. It will not be supported in 2.1 or higher. Do not write drivers using this obsolete technology.

5.1.8. Important Note

Within an interrupt driver the use of ThreadSleep, ThreadSleepQueued, ThreadDelay and MsgWait is strictly prohibited.

5.2. TLCS900 Interrupt Level

With mCAT 2.0x within the kernel only interrupts up to level 4 (4 included) are disabled in critical sections. MCAT interrupts must not use higher levels than 4.

LEVEL 6 should not be used for interrupt service routines (ISR) at all - this level is reserved for DMA operation. That is because the HDMA's of the TLCS900 H-CORE CPU's can be stopped by a DI operation. They operate always on level 6! To guarantee best DMA performance, disabling of level 6 must be minimized if it is necessary at all.

LEVEL 5 is reserved for QuickISR Interrupts. QuickISR MUST not use ***any other level***!

5.3. QuickISR

QuickISR's are used for quick, „overhead-free“ interrupt service. Typical applications for those QuickISRs are counter overflow counters in software etc. - short, high frequency interrupt handlers. A QuickISR can communicate with other interrupt handlers or mCAT Tasks via shared memory only! A QuickISR must never call a MsgSend... or ThreadSignal system call - or any other system call that may cause a task/thread switch.

5.3.1. QuickISR on Toshiba TLCS900 Platforms (mCAT 2.10)

Best language to write QuickISR's is assembler, because C will use the stack to much. If a QuickISR handler is written in C - the installation of an own stack frame is recommended!

It is possible to pass a previously defined arguments with the QucikISR call. In that case assembly is required anyway, because C cannot handle arguments to interrupt functions.

The QuickISR is attached using the new IntSetTrap call. Use IntSetTrap for QuickISR's and non-maskable interrupts only (SWIx, NMI, WatchDog).

It is forbidden to attach an ISR with LEVEL < 5 using IntSetTrap. Do not load the stack within a QuickISR with more than 8 bytes (2 32-bit values)

The two examples show a simple overflow counter. Example 1 uses a fixed memory location for the overflow counter. Example 2 uses a pointer to this counter passed as an argument!

```
overflow_counter    dl     1

quick_1:
      push         xwa                 ; save xwa
      ld           xwa,1               ; increment counter
      add          (overflow_counter),xwa
      pop          xwa
      reti                             ; MUST be a RETI!


quick_2:
      push         xwa                 ; save xwa
      ld           xwa,(xsp+4)         ; get pointer to argument
      ld           xwa,(xwa)           ; get pointer to counter
      addw         (xwa+),1            ; increment LSW
      adcw         (xwa),0             ; add carry to MSW
      pop          xwa                 ; restore xwa
      add          xsp,4               ; remove argument pointer
      reti                             ; MUST be a RETI!
```

5.3.2. QuickISR on ARM7 platforms (mCAT 2.10-R00168)

The major difference to the TLCS900 platforms, mCAT 2.10-R00168 does not need assembly-level programming for QucikISR's. The handler will look like this:

```
UINT32 counter;
void MyQickIsr(UINT32 *counter)
{
       *counter = *counter + 1;
}
       ...
       IntSetTrap(INT_LINE_3,MyQuickIsr,&counter);
       IntEnable(INT_TIMER_0,SYS_INT_LEVEL_5);
       ...
```

5.4. Relation with other mCAT concepts

Interrupt Drivers and Tasks are the major module types. From the outside, the interface is message passing and for a user it should not make any difference whether a service is implemented as a task or as an interrupt driver.

Interrupt drivers are used if the response time needed is below 5ms.

5.5. Function reference

The INT group offers functions to install and control interrupt handlers.

These functions are only supported on TLCS900 hardware.

IntInstall

Function: Installs a handler for hardware interrupts. The location of the workspace is assigned by mCAT, only the size must be specified.

C-Prototype: **INT *IntInstall (INTEGER intid, INTEGER (*service)(), INTEGER (*cservice)(), INTEGER (*wakeup)(), INTEGER (*notify)(), long stack-size, long wssize);**

Arguments:

intid Interrupt identifier

service Pointer to local function service. Used for assembly language code. Not supported on non TLCS platforms.

cservice Pointer to local function cservice. Used for C language code. If supplied, set service = NULL.

wakeup Pointer to function wakeup or NULL if no wakeup function is needed. To activate “PASSMSG” wakeup, use the marco “PASSMSG()” to pass wakeup.

notify Pointer to function notify or NULL if no notify function is needed. “notify” functions are obsolete and should no longer be used.

stacksize Size of private stack (not supported on all platforms)

wssize Size of private workspace. At least sizeof(WS)

Returns: Pointer to Int workspace

Supported:

mCAT	All versions. With mCAT2.08 the new “PASSMSG” wakeup call is available (see “wakeup”).
Hardware	All

Comments: *IntInstall with a “PASSMSG” type wakeup function:*

```
ws = IntInstall(  IntLine4,               // Interrupt line 4
                  NULL,                   // no assembler ISR
                  cisr,                   // C ISR!
                  PASSMSG(cwakeup),       // wakeup function
                  NULL,                   // no “notify”
                  0x100,                  // stacksize
                  sizeof(WS)              // minimum size of WS
                );
ws->imd = &my_imd; // insert link to IMD
```

IntSetTrap

Function:	Attaches the ISR „isr“ to interrupt „intid“ (see appendix a for a list of valid intid's). If „arg“ is not NULL, it will be passed on the top of stack to the ISR, else the stack is empty on ISR entry. If there was already an ISR attached, the pointer to the previously used ISR is returned.	
C-Prototype:	**void *IntSetTrap(INTEGER intid, void *isr, void *arg)**	
Arguments:	intid	Interrupt identifier
	isr	pointer to ISR
	arg	pointer to optional argument or NULL
Returns:		previously attached ISR or NULL
Supported:	mCAT	mCAT 2.06 and higher
	Hardware	All
Comments:		

IntEnable

Function:	Enables the specified interrupt source (TLCS900 INTE'xx' register)		
C-Prototype:	**INTEGER IntEnable (INTEGER intid);**		
Arguments:	intid	Interrupt identifier	
Returns:	Error Code	SYS_ERR_ILLEGAL_INT	No valid interrupt number
Supported:	mCAT	All versions	
	Hardware	All	
Comments:			

IntDisable

Function:	Disables the specified interrupt source.		
C-Prototype:	**INTEGER IntDisable (INTEGER intid);**		
Arguments:	intid	Interrupt identifier	
Returns:	Error Code	SYS_ERR_ILLEGAL_INT	No valid interrupt number
Supported:	mCAT	All versions	
	Hardware	All	
Comments:			

IntSetLevel

Function: Sets the level of the specified interrupt

C-Prototype: **INTEGER SetIntLevel (INTEGER intid, UNSIGNED level);**

Arguments:

intid	Interrupt identifier
level	Interrupt level

Returns:

SYS_ERR_ILLEGAL_INT number	No valid interrupt
SYS_ERR_ILLEGAL_LEVEL	No valid level number

Supported:

mCAT	All versions
Hardware	All

Comments:

IntGetLevel

Function: Reads the level of the specified interrupt

C-Prototype: **UNSIGNED IntGetLevel(INTEGER intid)**

Arguments:

intid	Interrupt identifier

Returns: Interrupt level

Supported:

mCAT	All versions
Hardware	All

Comments:

IntIsPending

Function: Checks whether there is a pending interrupt for the specified handler.

C-Prototype: **INTEGER IntIsPending (INTEGER intid, INTEGER *error);**

Arguments:

intid	Interrupt identifier	
error	Pointer to error variable:	
	SYS_ERR_OK	ok
	SYS_ERR_ILLEGAL_INT	No valid interrupt number

Returns: TRUE is interrupt is pending

Supported:

mCAT	All versions
Hardware	All

Comments:

6. mCAT Shared Libraries

6.1. The Concept of Shared Libraries

6.1.1. What is a Shared Library

A shared library is a piece of code implementing a set of functions that are not linked with the application at compile time but at runtime. Shared Libraries are helpful to reduce the code space needed, because the same library can be used from different modules.

In contrast to a message based interface (client-server), a Shared Library is a synchronous interface. It is not prepared to deal with multitasking. If you have a single resource to manage and concurrent access by several tasks you should not use Shared Libraries.

6.1.2. A Shared Library Call

The C-Runtime library (linked with every C-module) supports a small wrapper function called “__exec()”. This call is the one and only entry to the shared library mechanism. All Shared Library calls are implemented as C-Macros in the form:

```
#define MyLibraryFunction(p1,p2,..,pn)        __exec(0x40020l,p1,p2,..,pn)
```

The first number (here 0x40020l) is the Shared Library access code. It is a 32-Bit number formed of:

libid << 2		fid << 2	
31	16	15	0

“libid” is the libraries ordinal number assigned by “MIC”, the interface compiler. “fid” is the ordinal number of the function inside the Library “libid”.

6.1.3. Why Should I use “LOCAL” Structure?

MIC will presume that you will store all Library specific global variables into a special structure called “LOCAL”. This structure must be defined by the user (in the COMON/END-COMON section of the LDF-File).

The advantage is that LOCAL will be automatically allocated at startup. You will not have the need to worry where to place your globals at runtime. That way its role is a bit like ***WS*** in an interrupt driver. Every function called gets a pointer to LOCAL as an argument.

6.2. Relation with other mCAT Concepts

All mCAT API-Functions of all modules (NVMEM, MEM, SERDRV, ...) including the kernel itself are implemented as Shared Library

6.3. The Library Descriptor

Be aware that this data structure is documented for completeness and reference only! The structure should not be accessed directly! This structure is subject of change without notification!

```
typedef struct {
      IMD                 *imd;                 /* pointer to module IMD */
      UNSIGNED            libid;                /* ordinal number of the library */
      UNSIGNED            calls;                /* number of calls in jump table */
      INTEGER             (*func[1])();         /* jump table */
} LIB;
```

6.4. MIC & LDF-Files

The MIC (mocom interface compiler) is used to make Shared Library design easy. The library and all its functions are described in a so called LDF (LIBRARY DEFINITION FILE). MIC generates all files needed (modules initialization, wrapper files, includes). You just have to add your code and a makefile. For MIC and LDF file format please refer to “mCAT tools Documentation”.

6.5. Function Reference

This group adds some fundamental functions to the kernel. Most functions deal with the SHARED LIBRARY, the repository for all of mCATs code. The shared library can be extended by buying additional modules ore writing them yourself.

SysAddLib

Function: Installs a Shared Library (ShLib) with library number libid and a pointer lib to a library descriptor. This call is used in module initialization code generated by MIC.

C-Prototype: **INTEGER SysAddLib (UNSIGNED libid, void *lib, void *local);**

Arguments:

libid	the major number of the library to be installed
lib	pointer to the library descriptor
local	pointer to local memory for that library

Returns:

Supported:

mCAT	All versions
Hardware	-/-

Comments:

SysGetLib

Function: Returns a pointer to a ShLib descriptor. May be used for status monitoring or fast direct access to the functions.

C-Prototype: **void *SysGetLib (UNSIGNED libid);**

Arguments: libid the major number of the library to be installed

Returns: pointer to the library descriptor

Supported:

mCAT	All versions
Hardware	-/-

Comments:

7. mCAT Trace & Debug Interface

7.1. Function Reference

Currently, trace functions are under development. For now, only the TraceWriteLog command is available.

TraceWriteLog

Function: This function is used to write a string into the bootlog. The bootlog is used to document the boot process and runtime errors. It can be read using the command “show” of the SYSMON monitor.

User modules are allowed to write to the bootlog file too. Please note that strings are not copied to bootlog but just their references are saved.

C-Prototype: **void TraceWriteLog (char *msg);**

Arguments: msg Pointer to user ASCII-Z-string.

Returns: None

Supported:

mCAT	All versions
Hardware	All

Comments: Strings send to the bootlog by TraceWriteLog shall:

- start with a cr-fl sequence followed by a single space (“\r\n ”)

- end without any further cr and/or lf

8. mCAT Modules and the IMD (Initial Module Descriptor)

8.1. The Module / IMD Concept

As we heard before, a *module* is a compiled and linked unit of executable code. There are tasks with one or more threads, a kernel thread, an interrupt driver, a shared libary, just initialization code. A module can contain one, some or all of these concepts at a time. If a module is placed in (FLASH) ROM, mCAT will find it at system startup time and call its TaskInit function automatically.

An IMD (initial module descriptor) is needed in front of every module for automatic module detection. This is a special data structure containing all information needed to start and maintain the module.

Some of the information in the IMD is reserved for future use, some is used for task and interrupt drivers only (stack, heap).

The important information in the header include:

- The name of the module
- The version (Format: “#.##”) of the module
- The time tag “build” (UNIX time). The tag will be inserted by the utility TAG.EXE after the module was linked.

8.2. IMD Reference

```
typedef struct {
      word   pattern;            /* aa55=AUTOSTART, 0055, ff55 */
      void          (*init)();   /* pointer to module init function */
      void   (*main)();          /* pointer to task main */
      lword  stack;              /* stack size for this module */
      lword  heap;               /* heap size (currently not supported) */
      lword  build;              /* 32-Bit UNIX time stamp*/
      byte   priority;           /* initial priority */
      byte   mode;               /* mode, currently not used */
      byte   id;                 /* BITBUS function ID */
      char   version[5];         /* “X.XX” C-String */
      char   name[17];           /* “name” C-String */
      word   check;              /* hash of “name” */
} IMD;
```

8.2.1. mCAT 2.10-R00168

With mCAT 2.10-R00168 the IMD was extended to hold more information. This includes the memory layout and the reserved memory information. Both are retrieved and inserted into the IMD by the S3PATCH tool used in every standard mCAT makefile.

```
typedef struct {
    UINT32  base;       // start address
    UINT32  length;     // length of a memory area
} AREADESC;

typedef PACKED_STRUCTURE struct {
      UINT16        pattern;          // aa55=AUTOSTART, 0055, ff55
      INTEGER       (*init)();        // pointer auf init funktion
      INTEGER       (*main)();        // pointer auf task main function
      UINT32        stack;            // stack size
      UINT32        heap;             // reserved
      UINT32        build;            // build time
      UINT8         priority;         // start prio
      UINT8         mode;             // mode
      UINT8         id;               // BITBUS FID
      char          version[5];       // "x.xx"
      char          name[17];         // "name"
      UINT16        check;            // checksum (name)
      UINT16        exlen;            // length of extended IMD from here on!
      AREADESC      rom;              // rom address and length
      AREADESC      ram;              // ram address and length
      AREADESC      romres;           // reserved for rom address and length
      AREADESC      ramres;           // reserved for ram address and length
} GNU_PACKED_STRUCTURE IMD;
```

SysGetImdPtr

Function:	Returns a pointer to a tasks or shared libraries IMD structure. May be used for status monitoring.		
C-Prototype:	**IMD SysGetImdPtr (INTEGER cmd, INTEGER id);**		
Arguments:	cmd	SYS_GET_IMD_BY_LIBID	(0)
		SYS_GET_IMD_BY_ID	(-1)
		SYS_GET_IMD_BY_ACTIVE	(-2)
	id	taskid / libid / intid	
Returns:		Pointer to task, interrupt driver or shared library IMD.	
Supported:	mCAT	All versions, with mCAT2.10T00284 and later the three commands are supported. Before that version, the function returned the LIBIMD if cmd was 0 and the task/intid if it was not 0. The new command SYS_GET_IMD_BY_ACTIVE returns the IMD of the currently active task or interrupt driver, even at boot time!	
	Hardware	All	
Comments:			

8.3. INIT-Modules

8.3.1. What are INIT-Modules?

Init-Modules (aka Init) do not contain Task, Thread, Interrupt-Driver or Library creation code. Usually a Init-Module contains a TaskInit() function only.

The purpose of this type of functions is to modify the system at startup or at first use. The Function TaskInit() is executed once at system start.

Inits can be used very creatively. For example we use INITs to setup the EEPROM memory to default values when a hardware is powered up for the first time. This SYSTEM INIT will program the EEPROM and then mark the page it resides in for deletion. Finally, the INIT will use SysReset() to issue a restart.

So after the first (successful) powerup, the EEPROM is set, the INIT is removed and the hardware and software using the values in EEPROM is well initialized. For example, the BIT-BUS parameters (speed, node address, message length and number of buffers) are set to default (375kBit/s, 3, 255, 8).

Another idea would be an INIT that checks a given RAM area for a valid module. INITcan than start the RAM based module by calling its TaskInit() function.

> *The use of the functions ThreadSleep, ThreadSleepQueued, ThreadDelay and MsgWait is not allowed inside an INIT!*

8.3.2. How to Write an INIT

The makefile is pretty straight forward. The important switches to CIMD are:

```
-init=Init   Rename TaskInit to Init
-main=NULL   We need no TaskMain
-initmodule  We are composing a INIT module
```

here is the full source of makefile:

```
PROJECT = initdemo
TARGET = std_rom
CMD = -D$(CORE) -D$(HARD)
OBJFILES = imd.$(REL) $(PROJECT).$(REL)
INCFILES =

$(PROJECT).shx: $(OBJFILES)
	$(MKLNK) $(TARGET) $(PROJECT) $(OBJFILES)
	$(LD) $(PROJECT).LNK -o$(PROJECT).abs
	$(CONVERT) $(PROJECT).abs $(OF) $(PROJECT).shx
	$(S3PATCH) -l=$(PROJECT).map $(PROJECT).shx

$(PROJECT).$(REL):  $(PROJECT).c  $(INCFILES)
imd.$(REL):         imd.c
imd.c:              makefile
	$(CIMD) -auto -init=Init -main=NULL -initmodule -version=1.01 "mCAT/JustAInit" imd.c
```

The C-Source is easy also pretty easy to understand:

```
/*
 *    INITDEMO
 *
 *    (c) 1999, 2004 mocom software GmbH & Co KG
 *
 *    File: INITDEMO.C
 *
 *    History:
 *
 *     date        version author comment
 *    ----------------------------------------------------------------------
 *     08.04.1999  V1.00   VG     created
 *     06.06.2004  V1.01   VG     use CIMD, cross platform compatible
 *    ----------------------------------------------------------------------
 */
#include <mcat.h>
#include <simpleio.h>
#include <nvmem.h>
#include <eeprom.h>

long const purge = 0x0000;

void Init (IMD *imd)
{
    /* a little menu */
    loop {
        /* write out menu */
        WrStr("\n\n\n");
        WrStr("     1. START\n");
        WrStr("     2. RESET\n");
        WrStr("     3. REMOVE\n\n");
        WrStr("     YOUR CHOICE? ");

        /* wait for input, trigger watchdog meanwhile (TSM900 ONLY) */
        while (!kbhit()) {
#ifdef TSM_900
#include <t95c061.h>
    *P5 |= 0x04;
    *P5 &= 0xfb;
#endif
        } /* endif */

        /* get char and decide what to do */
        switch(RdChar()) {
          case '1':
            WrStr("START\n");   /* continue boot process, start mcat */
            return;

          case '2':
            WrStr("RESET\n");   /* reset system */
            SysReset();

          case '3':
            WrStr("REMOVE\n");  /* remove this little INITDEMO */

            FLASHWrite(imd,&purge,sizeof(purge));
            SysReset();

          default:
            WrStr("?\n");       /* UPS! */

        } /* endswitch */
    } /* endloop */
}
```

9. mCAT Miscellaneous System Functions

SysReset

Function: Forces a true hardware reset using the watchdog reset mechanism of the processors.

C-Prototype: **void SysReset (void);**

Arguments: -/-

Returns: -/-

Supported:

mCAT	All mCAT Versions
Hardware	All

Comments:

SysScan

Function: Scans the EPROM for initial module descriptors. As it finds one, it calls the user routine todo and continues. After memory is scanned it returns the number of found IMDs. Could be used to generate a list of available modules, for example.

C-Prototype: **Short SysScan (void *start, long length, INTEGER(*todo)());**

Arguments:

start	Start address for area to scan
length	Length of memory area to scan
todo	Pointer to a user written function that will receive a pointer to the found IMD as a parameter.

Returns: Number of IMDs found

Supported:

mCAT	All Versions
Hardware	All

Comments: If start == NULL, the std. Flash memory is scanned.

SysCalcHash

Function:	Calculates the hash code of a string (IMD.name)	
C-Prototype:	**UINT16 SysCalcHash (char *string);**	
Arguments:	string	Pointer to a string
Returns:		hash
Supported:	mCAT	All versions
	Hardware	All
Comments:		

10. The mCAT Ticker Sevice

10.1. What is the Ticker Good for?

Even if the Ticker is implemented in a separate module outside the kernel, its function is so basic for mCAT that it is included in the kernel reference.

The ticker is an interrupt service handler for the system timer. The usable resolution is 10ms[2] on Toshiba TLCS900 platforms and 1ms on ARM platforms. Within the ticker, two basically different services are maintained:

- An interval time based message responder (classical TICKER)
- A flexible software timer mechanism (ExpressTimer)

The message responder is one of the most frequently used services in mCAT. A task can instruct ticker to send a given message either after a given time or at a fixed frequency back to the task. This will make it easy to handle periodic intervals in a message driven task. There are two API calls needed to use the message responder service: TALL (previously ALL) and TAFTER (previously AFTER).

The message responder offers a fastest interval time of 10ms on the Toshiba TLCS900 platform and 1ms on the ARM platform.

10.2. TALL

[2] On TLCS900 platforms the ExpressTimer can operate at 5ms interval time in contrast to the 10ms available with normal Ticker service. On ARM, 1 ms is available with both concepts.

TALL

Function: Request the ticker to send this message back at a given interval rate. When the task receives a ticker message it can acknowledge the message using MsgSendReply(msg,Self,ACK). In this case the message will be scheduled for the next interval. The task can also not acknowledge the message using MsgSendReply(msg,Self,NAK), the ticker stops sending messages in this case.

C-Prototype: **MSGID *TALL (TickerMsg *msg, UINT32 ms, UNSIGNED priority, INTEGER self);**

Arguments:

msg	A private TickerMsg structure. This is the message used to notify your task about a elapsed interval.
ms	The interval time
priority	The priority that shall be used when sending ***msg***.
self	Your own task id.

Returns:

MSGID	Pointer to the message id structure used for ticker messages.

Supported:

mCAT	All versions
Hardware	All

Comments: In earlier mCAT system, this function was called ALL. The basic difference between TALL and ALL is that with TALL the MSGID of the ticker/all service is returned while with ALL the MSGID was not returned but stored in a global variable called TickerId. Both calls and mechanisms are still available on the TLCS900 platform. On the ARM platform, only TALL is available.

10.3. TAFTER

TAFTER

Function: Request the ticker to send this message back after a given interval. When the task receives a ticker message there is no reason to acknowlege it.

C-Prototype: **MSGID *TAFTER (TickerMsg *msg, UINT32 ms, UNSIGNED priority, INTEGER self);**

Arguments:

msg A private TickerMsg structure. This is the message used to notify your task about a elapsed interval.

ms The interval time

priority The priority that shall be used when sending ***msg***.

self Your own task id.

Returns: MSGID Pointer to the message id structure used for ticker messages.

Supported:

mCAT	All versions
Hardware	All

Comments: In earlier mCAT system, this function was called AFTER. The basic difference between TAFTER and AFTER is that with TAFTER the MSGID of the ticker/all service is returned while with AFTER the MSGID was not returned but stored in a global variable called TickerId. Both calls and mechanisms are still available on the TLC-S900 platform. On the ARM platform, only TAFTER is available.

10.4. The TickerMsg Structure

Usually the contents of the TickerMsg is of no interest to the user. However, it is not only documented for completeness. The member ***tag*** may be of interest sometimes. Tag is set to 0 with a TALL / TAFTER function is called or when a TickerMsg is acknowledged by the users task using *MsgSendReply()*. It is incremented whenever an interval period is completed. In that case, the message is only send to the usertask, if tag is 0 before its incremented. That means: If you receive a TickerMsg with tag > 1, than you missed to serve the message since tag-1 intervals! This can help to analyze and to control the responsiveness of the users task.

```
typedef struct {
      MSG          msg;        // mcat message header
      INTEGER      requester;  // caller
      INTEGER      prio;       // prio memory
      UNSIGNED      tag;         // status marker
      UNSIGNED      cmd;         // timer or ticker
      void          *internal;  // internal use only!
      XT            xt;          // placeholder for XT struct
} TickerMsg;
```

10.5. What is an ExpressTimer (XT)?

An ExpressTimer is a timer object used to implement fast, interrupt based timer services. Those timers should only be used carefully. Use as few as possible, best if you can avoid to use them. The code you can assign to those timers is executed in the timer interrupt handler context. It slows down the interrupt handling time and therefore the code should be as fast and straight forward as possible.

On the Toshiba TLCS900 platform the ExpressTimers time intervals can go down to 5ms. On the ARM platform the interval can go down to 1ms.

10.5.1. The ExpressTimer Handler Function

The prototype for an XT handler function is:

```
      INTEGER SYS_FDECL xt_handler(XT *self);
```

The function is called once every time the timers interval elapses. If it returns TRUE, the timer will be removed and no longer be serviced. To receive upcoming intervals, return FALSE.

The handler must be fast and lean. A user **MUST** not call ExpressIo functions from within an XT handler. Even if not forbidden, it is not recommended to send messages from within a handler. Never call MsgWait, ThreadSleep or ThreadDelay in an XT handler.

So what is an XT good for if all the good things are not allowed? Here is an example from ExpressIo. The Driver for the TSM8AD8 uses an XT driven state machine to read the (very slow) AD channels. The values read are stored in RAM for easy access. The state machine scans all channels by switching the multiplexer and starting the conversion. The state change is implemented by switching the handler function manually. The schedule will look like this:

t	***Action***
0	`xt_init, set MUXER = 0`
10	`xt_start_conversion`
20	`xt_read, data for channel 0, read values and set muxer to 1`
30	`xt_start_conversion`

t	*Action*
40	xt_read, data for channel 1, read values and set muxer to 2
50	...

And here is the code:

```
INTEGER xt_read (AD8XT *xt);                   // READ VALUES AND SET MUXER.
INTEGER xt_start_conversion (AD8XT *xt);       // START CONVERSION AFTER
                                               // MUXER SETTLED.

INTEGER xt_start_conversion (AD8XT *xt)
{
    // set next state
    xt->xt.code = xt_read;

    // start conversion
    outb(((UINT32)xt->ctrl.hdr.base)+2,0);

    return FALSE;
}

INTEGER xt_read (AD8XT *xt)
{
    UINT8   channel;

    // set next state
    xt->xt.code = xt_start_conversion;

    // get channel and increment
    channel = xt->ctrl.chan;
    if (channel >= 7) {
        xt->ctrl.chan = 0;
    } else {
        xt->ctrl.chan++;
    } /* endif */

    // read data for present channel
    xt->ctrl.data[channel] = inb((UINT32)xt->ctrl.hdr.base);

    // set muxer to next channel
    outb((UINT32)xt->ctrl.hdr.base,(UINT8)xt->ctrl.chan);

    return FALSE;
}

// USED TO INIT THE STATEMACHINE
INTEGER xt_init (AD8XT *xt)
{
    UNSIGNED i;

    // set basic state
    xt->xt.code = xt_start_conversion;
    // preset data
    for (i=0;i<8;i++) {
        xt->ctrl.data[i] = 0;
    } /* endfor */

    // set muxer to channel 0
    xt->ctrl.chan = 0;
    outb((UINT32)xt->ctrl.hdr.base,(UINT8)xt->ctrl.chan);

    // iknstall ExpressTimer, using already set handler
    XTAdd(&xt->xt,10,xt->xt.code);

    return FALSE;
}
```

10.5.2. The ExpressTimer Data Structure

Usually there should be no reason to access this structure directly. Use API functions instead. However, there is one exception: Changing the handler manually (member ***code***, see example in previous chapter). Please also note that the layout and use of the members are slightly different on the Toshiba TLCS900 platform and the ARM platform.

```
typedef struct __xt__ {
      struct __xt__       *next;      // next in ticker chain
      struct __xt__       *prev;      // prev in ticker chain
      UINT32              schedule;   // schedule in ms
      UINT32              counter;    // down counter
      INTEGER             cmd;        // command code
      INTEGER             (*code)();  // handler
} XT;
```

10.5.3. The XT-API functions

XTAddTimer

Function: Adds a timer that is removed unconditionally after the interval time has elapsed.

C-Prototype: **void XTAddTimer (XT *xt, UINT32 time, INTEGER (*code)());**

Arguments:

xt	Pointer to an XT timer structure
time	The interval time
code	Pointer to handler function

Returns: -/-

Supported:

mCAT	All versions
Hardware	All

Comments:

XTAdd

Function: Adds a timer that is called periodically whenever the interval time elapsed and until the handler function returns TRUE.

C-Prototype: **void XTAdd (XT *xt, UINT32 time, INTEGER (*code)());**

Arguments:

xt	Pointer to an XT timer structure
time	The interval time
code	Pointer to handler function

Returns: -/-

Supported:

mCAT	All versions
Hardware	All

Comments:

XTRemove

Function: Remove an XT from the handling queue unconditionally.

C-Prototype: **void XTRemove (XT *xt);**

Arguments:

xt	Pointer to a XT timer structure

Returns: -/-

Supported:

mCAT	All versions
Hardware	All

Comments:

XTSet

Function: Set a new interval time and handler function.

C-Prototype: **void XTSet(XT *xt, UINT32 time, INTEGER (*code)());**

Arguments: xt Pointer to a XT timer structure

Returns: -/-

Supported:

mCAT	All versions
Hardware	All

Comments:

XTGetResolution

Function: Returns the resolution of the ExpressTimer System. On TLCS900 platforms this will be 5ms, on ARM platforms it will be 1ms.

C-Prototype: **UNSIGNED XTGetResolution(void);**

Arguments: string Pointer to a string

Returns: hash

Supported:

mCAT	All versions
Hardware	All

Comments:

11. Error Code Cross Reference

Porting mCAT to the ARM platform was an opportunity to change the existing naming conventions to follow more common standards. The table below shows old and new error codes. Users should use the new style constants only! Please note that the old error codes are still available for compatibility.

Old Style Error Codes	*New Style Error Codes*	*Numeric Value*
ErrOk	SYS_ERR_OK	0
ErrNil	SYS_ERR_NIL	1
ErrRun	SYS_ERR_RUNNING	2

Old Style Error Codes	***New Style Error Codes***	***Numeric Value***
ErrAllInUse	SYS_ERR_ALL_IN_USE	3
ErrNoRam	SYS_ERR_OUT_OF_MEMORY	4
ErrNoTask	SYS_ERR_NO_TASK	5
ErrNotSuspended	SYS_ERR_NOT_SUSPENDED	6
ErrTrap	Not used	7
ErrNoTarget	SYS_ERR_NO_TARGET	8
ErrNotSelf	Not used	9
ErrNoEvent	SYS_ERR_NO_EVENT	10
ErrNoFreeMem	Duplicat of ErrNoRam	11
ErrIsSuspended	SYS_ERR_IS_SUSPENDED	12
ErrNoKernelFunction	SYS_ERR_UNKNOWN_FUNCTION	13
ErrIdNotFound	SYS_ERR_ID_NOT_FOUND	14
ErrIdOverflow	SYS_ERR_ID_OVERFLOW	15
ErrNoValidLevel	SYS_ERR_ILLEGAL_LEVEL	16
ErrWrongIntNo	SYS_ERR_ILLEGAL_INT	17
ErrNotFound	Duplicat of ErrIdNotFound	18
ErrIdInUse	SYS_ERR_ID_IN_USE	19
ErrNotImplemented	SYS_ERR_NOT_IMPLEMENTED	20
ErrMsgNotSupported	SYS_ERR_NOT_SUPPORTED	21
ErrTerminated	SYS_ERR_TERMINATED	22
ErrTimeOut	SYS_ERR_TIME_OUT	23
ErrLocked	SYS_ERR_LOCKED	24
ErrArgOutOfRange	SYS_ERR_INVALID_ARGUMENT	25
ErrRttiCheckFailed	SYS_ERR_RTTI_CHECK_FAILED	26
ErrNoHeap	SYS_ERR_NO_HEAP	27
New Error Code	SYS_ERR_NO_THREAD	28
New Error Code	SYS_ERR_NO_QUEUE	29
New Error Code	SYS_ERR_FATAL	30
New Error Code	SYS_ERR_INTERNAL	31
New Error Code	SYS_ERR_SHIB_NO_LIBRARY	32
New Error Code	SYS_ERR_SHIB_NO_FUNCTION	33

V. mCAT ExpressIO^{TM}

1. Introducing ExpressIO^{TM}

1.1. Overview

ExpressIO^{TM} is a programmers interface designed to map physical process i/o to a logical layer that makes it easy to split the *configuration* off from the *logic* of a control application.

The *configuration* of a control application handles all the physical to logical mapping and the setup of an i/o characteristic. A physical i/o may be a single bit in an io-port register of your control system. A logical i/o should, in contrast, be usable without knowing physical details. Its name should also reference to its suggested functionality (“*conveyor_belt_moves”*) rather than to its physical location (“*in_1_1_2*”);.

From the *logics* point of view, there is no physical i/o: there are only IOOBJECTs representing the i/o.

One of the biggest advantages of splitting the configuration from the logic of an application, is that such an application is easier to port from one hardware to another. If an application runs fine on one hardware, usually only the configuration has to be changed to port it to another one. Typical case: You designed an application on one hardware and then a new hardware gets available that is cheaper and faster or has some other features you like to use. If both systems support ExpressIO^{TM}, porting your application from one system to the other is simple.

Another big advantage of ExpressIO^{TM} is its ability to add features to physical i/o ports the original hardware does not support. For example: It is possible to insert a so-called ExpressProgram, some kind of virtual i/o device, between a physical i/o and its logical representation. That can be an EDGE detector ExpressProgram, that periodically checks a simple, naked i/o port and signals if the value of this i/o changes. Using ExpressPrograms makes the application design smarter.

With ExpressIO^{TM}, the logical representation of an i/o port is known as an IOOBJECT.

1.2. A Note on Datatypes

With mCAT 2.20 we had to change a few data types and names of data types for better compatibility. However, the old mCAT 2.10 data types are still available and fully valid. Please take the following table as a reference for the different types:

MCAT 2.10	***MCAT 2.20***	***minimum***	***maximum***	***use***
byte	UINT8	0	255	
word	UINT16	0	65535	
lword	UINT32	0	4294967295	
-	UINT64	0	2^{64}-1	Not available with TLCS900
-	INT8	-128	127	
short	INT16	-32768	32767	
long	INT32	-2147483648	2147483647	
-	INT64	-2^{63}	2^{63}-1	Not available with TLCS900
int	INTEGER	-32768*	32767*	Default type for small integers
unsigned	UNSIGNED	0	65535*	Default type for unsigned integers
bool	BOOL	TRUE	FALSE	

Table 1: mCAT 2.20 data types

* The range of those types depend on the target platform. We assume that for those types the *16-bit* data range is the minimum we can rely on.

2. IOOBJECTs

The logical layer objects accessed by the *logic* of an application are called IOOBEJCTs.

The IOOBEJCTs (data type IOOBJECT) must be declared as global variables outside any function bodies. Because the C runtime startup code will zero out all global variables, by declaring them we just reserved memory to store them and gave them *compile time names*. To make IOOBJECT's usable, we have to initialise them. The process of initialization is called *to create an IOOBJECT.* The function *IOObjCreate()* does the job. It takes several arguments to make a logical connection between a physical input, an optional *ExpressProgram,* an optional *runtime name* and the IOOBJECT variable.

A *compile time name* is a name that is used by C to address an object, like a variable, a function or a constant value. These names are dropped in the compilation process and its important to understand that those names are not available at runtime! To provide *runtime names*, the *IOObjCreate()* call accepts an optional argument *name* where you can assign a visible string. If not needed,simply pass NULL instead of a string. *Please note that if you do not supply runtime name, your IOOBJECTS are not visible to mocom's OPC-Server or to the SYSMON ExpressIO™ functions.*

There is a small but efficient set of functions (or *methods* to keep the object orientated nomenclature) to access and configure IOOBJECTs.

2.1. Mapping Physical Ports to IOOBJECTS

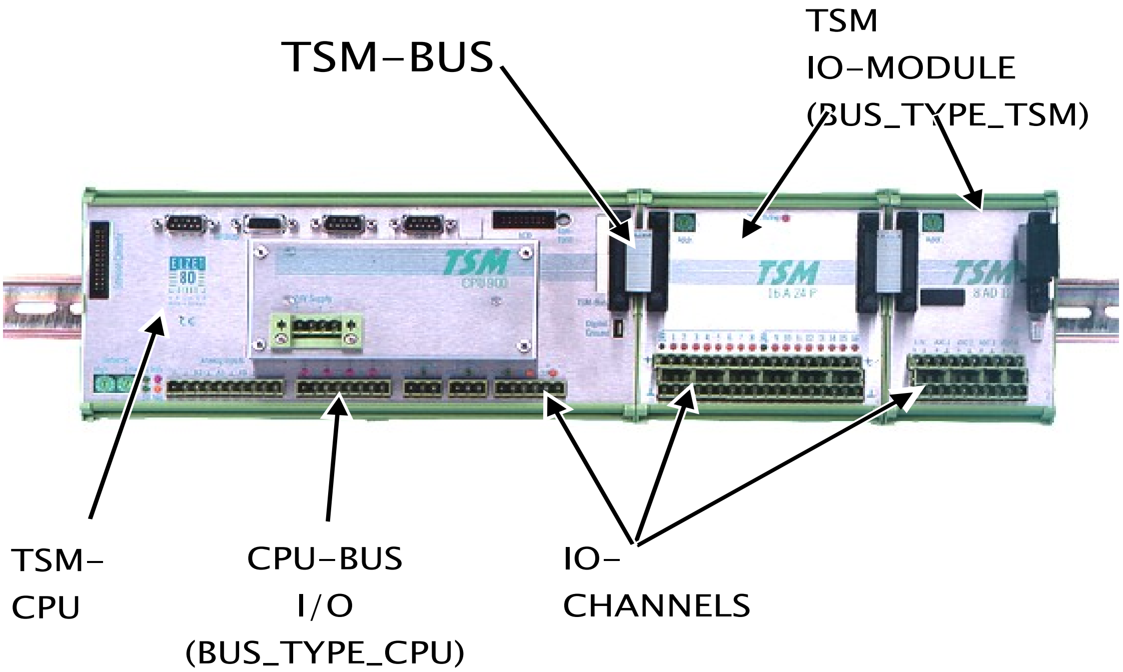


Figure 15: Example Hardware TSMCPU900 with TSM-Bus modules attached

2.1.1. ExpressIO^{TM} Physical Drivers

For every supported piece of hardware there must be a physical device driver, called an ***Express Driver*** or simply a *driver*. The driver abstracts the hardware to a common model of the specific hardware. The driver is the basic link between hard- and software.

2.1.2. Bus, Module, Channel: Referring to the Hardware

The IOObjCreate() call takes – beside others – three arguments that build the physical reference that is used to map the physical i/o to an IOOBEJCT. The base idea is that every physical port can be addressed by:

1. The ***bus*** it is attached to
2. The ***module*** address that identify it within the bus
3. The ***channel*** number that identify it within the module

The *bus* selector is usually a system defined constant. Typical constants are **BUS_TYPE_CPU**, **BUS_TYPE_I2C**, **BUS_TYPE_TSM**.

The *module* address is usually an integer value and hardware / bus specific. With TSM it is the TSM module address set via hex switches. For IO attached directly to the CPU, some enumerated values are provided to address the modules. See I.4. Supported Hardware Reference for more details.

The *channel* number is an integer value, starting with 0!

2.1.3. Classes

The CLASS identifier of an driver can be read by use of a "INFO" type call (see 2.4.). CLASS is a 16-Bit unsigned integer. Its contents are created by a bitwise OR operation of several possible values. The classes a driver supports can be retrieved by an AND operation of a drivers class value and these constant definitions. The definitions available are:

CLASS_INPUT

The device or channel is an *INPUT* device.

CLASS_OUTPUT

The device or channel is an *OUTPUT* device.

CLASS_DIGITAL

The device or channel is a *digital* IN- or OUTPUT. There is no information provided on the physical parameters like voltage, current or isolation. The class value simply describes the logical behavior.

CLASS_ANALOG

The device or channel is an *analog* IN- or OUTPUT.

CLASS_PWM

The device or channel is a *pulse width modulator* OUTPUT.

CLASS_FREQ

The device or channel is a *frequency counter* INPUT.

CLASS_EVTCNT

The device or channel is an *event counter* INPUT.

CLASS_POS

The device or channel is a *position encoder* INPUT. No information is provided whether it is an absolute or incremental position encoder.

2.1.4. The IOObjCreate Function

IOObjCreate

Function: Create an IOOBJECT by resolving the address information and bind it to a driver, an ExpressProgram and a runtime available name .

C-Prototype:

```
INTEGER IOObjCreate(IOOBECT *obj, char *name, UNSIGNED bus, UN-
SIGNED module, UNSIGNED chan, UNSIGNED class, char *xp);
```

Arguments:

	A pointer to a variable of type »IOOBJECT« which refers the created object.
name	An optional ASCII-name for this object
bus	The bus the I/O is connected to. Use predefined constants from xpconst.h here ONLY. They start with BUS_TYPE_...
module	Refers to the module that should be used
chan	Refers to the channel of this module
class	required I/O class.
xp	pointer to the name of the requested ExpressProgram

Returns:

IOERR_CREATE_FAILED if *obj* is a NULL pointer

IOERR_OBJECT_EXISTS if an object with the same *name* exists.

IOERR_NO_DRIVER if no physical driver to satisfy *bus,module,channel* was found.

IOERR_NOTEMPLATE *xp* was given but it could not be found in the database. Usually a spelling error.

IOERR_OUT_OF_MEMORY system runs out of memory while creating the IOOBJECT.

IOERR_OK if IOOBJECT was created successfully.

Comments:

2.1.5. The SYSTEM Function

To get the benefit from ExpressIO^{TM}, creation and configuration of IOOBEJCTs should be placed in a single function. The recommended name for this function is SYSTEM().

The benefit in gathering all the create and configuration stuff in a single function is that it makes it very easy to understand, change and enhance the configuration an application needs.

2.2. Vector Access versus Single Channel Access

To access an IOOBJECT, there are to classes of methods: Vector access and single channel access.

1. The vector access methods (***VECIN()*** and ***VECOUT()***) allow to read all i/o of a given hardware module with a single call. The IOOBJECT is used to address the given module only.
2. The single channel access methods (***IN()*** and ***OUT()***) really access the IOOBJECT itself.

So what type of access to use?

To get the most benefit of of ExpressIO^{TM}, use single channel access only. The vector access functions are far less portable and offer only a little performance advantage.

However, for some purposes these calls are the best (and only) choice. One example is the VECCOUNT ExpressProgram.

To use the vector mode, create one IOOBJECT that refers to modules channel 0 only.

2.3. The IOOBJECT Methods in Detail

IN

Function:	Read the value of an IOOBJECT	
C-Prototype:	`INT32 IN(IOOBJECT *obj);`	
Arguments:	obj	Pointer to the referred IOOBECT
Returns:		Value of the IOOBJECT.
Comments:	*Note: For digital i/o 0 or 1 is returned respectively.*	

OUT

Function: Write a value to a IOOBJECT

C-Prototype:

```
void OUT(IOOBJECT *obj, INT32 value);
```

Arguments:

obj	Pointer to the referred IOOBECT
value	The value to be written.

Returns: -/-

Comments: *Note: For digital i/o every value that is not 0 will set the object to 1.*

VECIN

Function: Read channels of the IOOBJECT's under laying module

C-Prototype:

```
int VECIN(IOOBJECT *obj, INT32 *data, UNSIGNED len);
```

Arguments:

obj	Pointer to the referred IOOBECT
data	Pointer to an array of integers to store the data read
len	Size of the data array in byte. Use the standard C sizeof operator to calculate len.

Returns: Number of bytes read. Divide this value by sizeof(INT32) to get the number of data items read.

Comments: *Note: With digital i/o this function returns a bitmap instead of an array of INT32 values.*

VECOUT

Function: Write values to the channels of the IOOBJECT's under laying module

C-Prototype:

```
int VECOUT(IOOBJECT *obj, INT32 *data, UNSIGNED len);
```

Arguments:

obj	Pointer to the referred IOOBECT
data	Pointer to an array of integers that shall be written to the module
len	Size of the data array in byte. Use the standard C sizeof operator to calculate len.

Returns: Number of bytes read. Divide this value by sizeof(INT32) to get the number of data items read.

Comments: *Note: With digital i/o this function returns a bitmap instead of an array of INT32 values.*

2.4. Configuration and Information Retrieval

INFO

Function: Retrieve the configuration of an IOOBJECT

C-Prototype:

```
INT32 INFO(IOOBJECT *obj, UNSIGNED cmd, UNSIGEND param);
```

Arguments:

obj	Pointer to the referred IOOBECT
cmd	Specifies the particularly configuration parameter to be read. The constant values that are allowed here can be found in XPCONST.H and start with INFO_GET_
param	This argument is reserved

Returns: The configuration value read or 0 if nothing read

Comments:

CFG

Function: Set the configuration of an IOOBJECT

C-Prototype:

```
INT32 CFG(IOOBJECT *obj, UNSIGNED cmd, INT32 value);
```

Arguments:

obj	Pointer to the referred IOOBECT
cmd	Specifies the particularly configuration parameter to be written to. The constant values that are allowed here can be found in XPCONST.H and start with CFG_SET_
value	The value to be set.

Returns: TRUE on success

Comments:

INFOCH

Function: Retrieve the configuration of an IOOBJECT. This method is used if a IOOBJECT is used to address a module for use with vector i/o methods. In this cases it might be necessary to still configure a module on a channel by channel base.

C-Prototype:

```
INT32 INFOIN(IOOBJECT *obj, UNSIGNED cmd, UNSIGNED ch, UNSIGEND param);
```

Arguments:

obj Pointer to the referred IOOBECT

cmd Specifies the particularly configuration parameter to be read. The constant values that are allowed here can be found in XPCONST.H and start with INFO_GET_

ch Number of the channel to access

param This argument is reserved

Returns: The configuration value read or 0 if nothing read

Comments:

CFGCH

Function: Set the configuration of an IOOBJECT. This method is used if a IOOBJECT is used to address a module for use with vector i/o methods. In this cases it might be necessary to still configure a module on a channel by channel base.

C-Prototype:

```
INT32 CFGIN(IOOBJECT *obj, UNSIGNED cmd, UNSIGNED ch, INT32 value);
```

Arguments:

obj Pointer to the referred IOOBECT

cmd Specifies the particularly configuration parameter to be written to. The constant values that are allowed here can be found in XPCONST.H and start with CFG_SET_

ch Number of the channel to access

value The value to be set.

Returns: TRUE on success

Comments:

2.5. “Express Programs”

An ExpressProgram is a virtual device that can be used like any physical device. It is used to add functionality to a physical device the physical device does not support. This may be a simple filter function or an edge detector. If the ExpressProgram generates i/o events – like **IO_EVT_FALL** (a falling edge was detected) – we need additional functions and methods to get informed about those events. But before we discuss the two interfaces available to handle i/o events, we should have a short look at the diagram below. It shows how an *ExpressProgram* is linked between an *IOOBJECT* and a *physical device*.

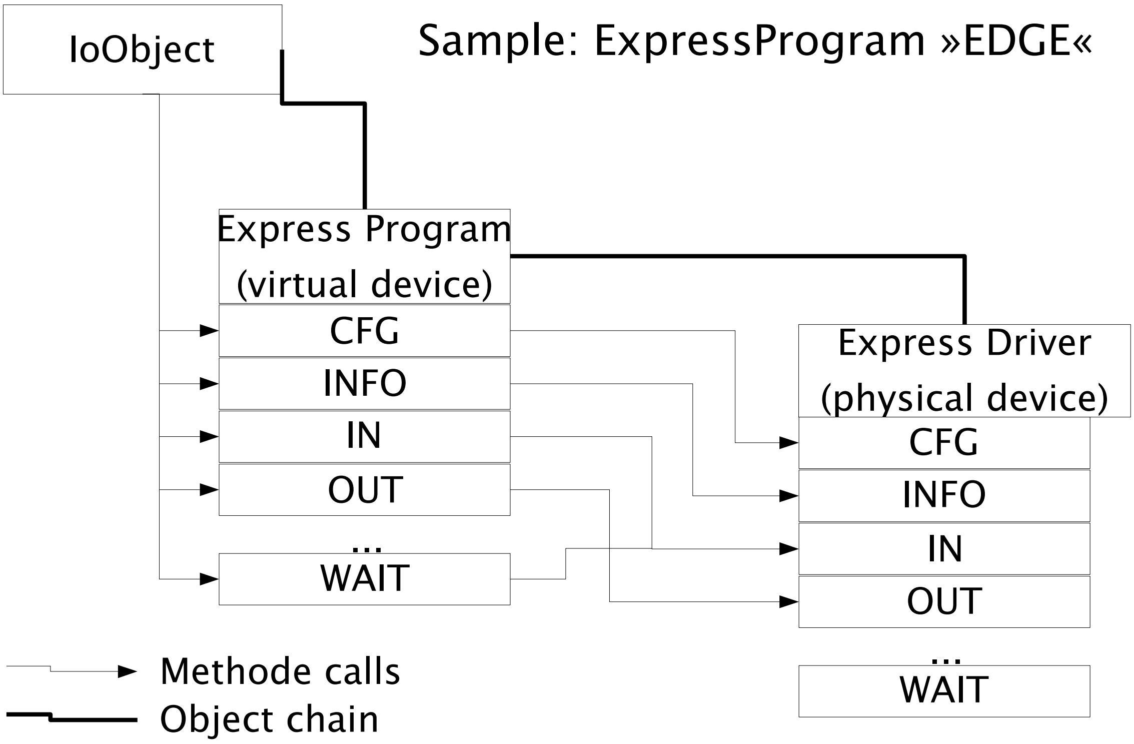


Figure 16: ExpressPrograms, linked between driver and IOOBJECT

2.5.1. The WAIT Interface

The WAIT interface supplies two functions that suspend execution of a program and wait for an i/o event to occur on a specific IOOBJECT.

NOTE: WAIT was designed to support non-message based applications and its use is only of limited use in the highly message based mCAT environment. Please take a look on the Subscription interface which is a more suitable interface for the same purpose.

WAIT

Function: Wait for a specific i/o event to occur. While waiting, the calling thread does not consume CPU time. This call makes sense when used with IOOBJECTS with attached ExpressPrograms (like EDGE or CAPTURE) only.

C-Prototype:

```
INTEGER WAIT(IOOBJECT *obj,UNSIGNED trigger,UINT32 timeout);
```

Arguments:

obj Pointer to the referred IOOBECT

trigger _RISE, _EDGE, _BOTH are the values used in previous versions of ExpressIO™. Please use the new values IO_EVT_FALL, IO_EVT_RISE, IO_EVT_BOTH, IO_EVT_ONE, IO_EVT_ZERO.

Note that _RISE, _EDGE, _BOTH are binary compatible with the new values, so a system compiles with :RISE instead of IO_EVT_RISE will still operate correctly.

timeout A timeout in milliseconds. If the WAIT call is pending for timeout ms without that trigger is matched, the call will be terminated and the calling process will be scheduled for execution.

Returns: 0 if fails, -1 if timeout occurred or any positive value on success.

Comments:

WAITIO

Function: Wait for a specific i/o event to occur. While waiting, the calling thread does not consume CPU time. This call makes sense when used with IOOBJECTS with attached ExpressPrograms (like EDGE or CAPTURE) only. WAITIO captures automatically the value of the IOOBJECT on trigger.

C-Prototype:

```
INTEGER WAITIO(IOOBJECT *obj,UNSIGNED trigger,UINT32 timeout,
INT32 data, UNSIGNED vlen);
```

Arguments:

obj Pointer to the referred IOOBECT

trigger _RISE, _EDGE, _BOTH are the values used in previous versions of ExpressIO™. Please use the new values IO_EVT_FALL, IO_EVT_RISE, IO_EVT_BOTH, IO_EVT_ONE, IO_EVT_ZERO.

Note that _RISE, _EDGE, _BOTH are binary compatible with the new values, so a system compiles with :RISE instead of IO_EVT_RISE will still operate correctly.

timeout A timeout in milliseconds. If the WAIT call is pending for timeout ms without that trigger is matched, the call will be terminated and the calling process will be scheduled for execution.

data Pointer to a array of INT32 values to store the captured data.

vlen Number of entries in the array *data.*

Please note this is not the size in bytes as with the VECIN & VECOUT calls!

Returns: 0 if fails, -1 if timeout occurred or the number of INT32 items read to the array *data.*

Comments:

2.5.2. Message Passing Interface

In an application that uses mCAT's highly event driven model, its much easier to handle additional messages that carrie information on i/o events than calling *WAIT* or *WIATIO*.

We use a *subscriber-consumer* model to implement a message based interface. Following this model, an application can subscribe for specific i/o-events on specific *IOOBJECTS.* The message structure used is ***XPEvent***. Only a few calls are needed to use this interface.

Refer to the example in the *cc\express\belt* directory of your mCAT installation to get an impression what is possible with our event driven interface.

2.5.2.1. Message Passing Interface – Function Reference

XPEventSubscribe

Function: Initialize an XPEvent and send it to the ExpressIO^{TM} XPServer. The XPServer maintains the ExpressPrograms installed. When the requested i/o event occurs, the XPEvent message is send back to the application to keep it informed.

C-Prototype:

```
MSGID *XPEventSubscribe(XPEvent *evt,UNSIGNED evtid,IOOBJECT
*obj,UNSIGNED trigger,UINT32 timeout,INT32 *value,UNSIGNED
vsize);
```

Arguments:

evt — Refers to an XPEvent message. This message is derived from a standard mCAT-Message. The access to the information carried by this message is available through the API. There is no need to directly refer to members of this message type.

evtid — This is an user supplied integer that is returned with event when its triggered. This id helps an user to sort out different XPEvents when using more than one Subscription.

obj — A reference to the IOOBJECT

trigger — An event to subscribe to:
IO_EVT_FALL if a falling edge is detected
IO_EVT_RISE if a rising edge is detected
IO_EVT_BOTH if an edge is detected
IO_EVT_ONE if input is 1
IO_EVT_ZERO if input is 0

timeout — An optional timeout value. Set to SYS_WAIT_INFINITE to wait infinitely.

value — A pointer to an array of INT32 to store the captured values from the device. These values are captured along with the detected event. If NULL, no data is captured

vsize — Size of the array *value* in items. Example:
INT32 my_data[4];
vsize = 4;

Returns: MSGID used to identify an XPEvent. NULL if Subcription fails.

Comments:

XPEventRenew

Function: If an XPEvent was triggered and received by the application, the XPEventRenew call extracts all information needed from the XPEvent and sends it back to the XPServer to be ready to catch the next event.

C-Prototype:

```
UNSIGNED XPEventRenew(XPEvent *evt, UNSIGNED *captured, UNSIGNED
*evtid);
```

Arguments:

evt Pointer to the received XPEvent

captured Pointer to a UNSIGNED integer. If provided, the number of ITEMS written to the users data array (the *value* argument in XPEventSubscribe). Maybe NULL.

id Pointer to an UNSIGNED integer. If provided, the id of the event is stored at this location (the *evtid* argument in XPEventSubscribe).

Returns:

IO_EVT_ERROR if an error occurred
IO_EVT_TIMEOUT if a timeout occurred
IO_EVT_FALL if a falling edge is detected
IO_EVT_RISE if a rising edge is detected
IO_EVT_BOTH if a edge is detected
IO_EVT_ONE if input is 1
IO_EVT_ZERO if input is 0

Comments:

XPEventTerminate

Function: If an XPEvent was triggered and received by the application, the XPEventTerminate call extracts all information needed from the XPEvent. It does not send it back and the Subscription terminates.

C-Prototype:

```
UNSIGNED XPEventTerminate(XPEvent *evt, UNSIGNED *captured, UN-
SIGNED *evtid);
```

Arguments:

evt	Pointer to the received XPEvent
captured	Pointer to an UNSIGNED integer. If provided, the number of ITEMS written to the users data array is stored at this location (the *value* argument in XPEventSubscribe). Maybe NULL.
id	Pointer to an UNSIGNED integer. If provided, the id of the event is stored at this location (the *evtid* argument in XPEventSubscribe).

Returns:

IO_EVT_ERROR if an error occurred
IO_EVT_TIMEOUT if a timeout occurred
IO_EVT_FALL if a falling edge is detected
IO_EVT_RISE if a rising edge is detected
IO_EVT_BOTH if a edge is detected
IO_EVT_ONE if input is 1
IO_EVT_ZERO if input is 0

Comments:

2.6. WatchDog Handling

A call to this function will update ALL outputs. On some systems, like ELZET80 TSM, all outputs are controlled by watchdog timers. It must be guaranteed that all outputs are frequently written. If this can not be guaranteed by the application normal i/o activities, a regular call to *DrvTriggerWD()* will make sure that the watchdogs are satisfied.

DrvTriggerWD

Function: Update all output modules without changing their value.

C-Prototype:

```
void DrvTrggerWD(void);
```

Arguments: -/-

Returns: -/-

Comments:

3. Putting it together: A Quick Start Tutorial

Lets assume a simple application. We have an input that signals whether a conveyor belt is moving or not. This input we name *belt_moves*. We have a digital output that switches an alarm horn. This we name *belt_fail_alarm*. We have to monitor *belt_moves* and if it does not move any more, we have to blow the horn by setting *belt_fail_alarm*.

First Step: Add The ExpressIO^{TM} Include File (<xio\express.h>) to the list of file to be included.

Second Step: Declare the i/o-objects needed. The i/o-objects, data type IOOBJECT, must be declared as global variables outside any function bodies. In our little application we have:

IOOBECT *belt_fail_alarm, belt_moves;*

Third step: Next step is to initialise the IOOBJECT's by calling IOObjCreate(). The *belt_moves* input is presumed to be connected to physical input 7 of TSM-Bus digital input module at TSM address 1. The *belt_fail_alarm* output is connected to physical output 2 of TSM-Bus digital input module at TSM address 3.

```
IOObjCreate(&belt_moves,
              NULL,BUS_TYPE_TSM,1,7,CLASS_DIGITAL,NULL);
IOObjCreate(&belt_fail_alarm,
              NULL,BUS_TYPE_TSM,3,2,CLASS_DIGITAL,NULL);
```

It is recommended to check the return value of IOObjectCreate. If everything is fine, it will be **IOERR_OK**. In any other case, an error occurred and the IOOBJECT was not created.

Fourth step: Now the objects are ready to use. The *IN(<ioobject>)* and *OUT(<ioobject>,<value>)* calls can be used to read and write the i/o. Assuming we have a function that is called periodically to control the belt, the code in our little example may look like this:

```
if (IN(&belt_moves) == 0) {
       OUT(&belt_fail_alarm,1);
}
```

Fifth step: If we want ExpressIO^{TM} to control whether the *belt_moves* input changed, we have to modify our little example. First, we have to create an EDGE detector ExpressProgram between the physical i/o device and our IOOBEJCT. Therefore we just have to change one parameter of our IOObjCreate call:

```
IOObjCreate(&belt_moves,
                NULL,BUS_TYPE_TSM,1,7,CLASS_DIGITAL,
                "EDGE");
```

Sixth step: Now we can subscribe to one or more events generated by the newly created IOOBJECT. Because we should alarm if the *belt_moves* input is 0, we subscribe for trigger event **IO_EVT_ZERO**. A timeout is not needed and we are not interested in any actual data of this input (because it is signalled implicitly).

```
XPEvent xpevt;
XPEventSubscribe ( &xpevt, 1, &belt_moves, IO_EVT_ZERO,
                        SYS_WAIT_INIFINITE, NULL,0);
```

Seventh step: We have to handle the event in our central message loop. For now, we just skip the details of a message loop and the message handling here. You can find the full source code of this example in the *cc\express\belt* directory of your mCAT installation.

The code is simple and straight forward. After we detect reception of an XPEvent, we sort out whether this is ours and signal failure if it is. Finally we have to renew our subscription.

Please note that in the full example code we use two inputs to show how to use the event id to efficiently distinguish between different i/o events.

```
} else if (msg->type == xpeventid->id) {
        // is this our belt_moves event? Then XPEventGetId() = 1
        if (XPEventGetId((XPEvent *)msg) == 1) {
                // yes, belt_moves is ZERO!
                OUT(&belt_fail_alarm,1);
        } else if (...) {
                // some other events to handle
        } else {
        } /* endif */
        XPEventRenew((XPEvent *)msg);
} else if (
        // some other messages to handle
```

4. ExpressIO^{TM} Reference

4.1. Using CFG and INFO Method Calls to Configure Devices

The standard methods ***CFG, CFGCH, INFO*** and ***INFOCH*** (see I.2.3. The IOOBJECT methods in detail) can be used to access a variety of configuration items. This chapter informs on the items available. The general use is always the same. The items name has to be passed in the configuration calls for the argument ***cmd***. Items that can be retrieved start with *INFO_GET_* and must be used by *INFO* and *INFOCH* calls only. Items that can be changed start with *CFG_SET_* and must be used by *CFG* and *CFGCH* calls only.

4.1.1. Basic Info and Configuration Calls

4.1.1.1. Hardware Module Identification

INFO_GET_IDENT

Returns the module identification code. TSM modules will return their ID, TSMCPU devices will return their ordinal module number.

param: Not used (0)

return: Module ID

INFO_GET_IDENT_STRING

Returns a pointer to a string. The string contains the module identification as readable text.

param: Not used (0)

return: pointer to an ASCIIZ string. Note that the return value of INFO and INFOCH is an 32-Bit-Integer. It must be casted to prevent compile time errors by the C compiler.

example:

```
char *name;         /* we assume an IOOBJECT »my_ioobject« is declared */
INT32 id;

name = (char *) INFO(&my_ioobject, INFO_GET_IDENT_STRING, 0);
if (name) {
      id = INFO(&my_ioobject, INFO_GET_IDENT, 0);
      printf(string,"Module '%s' [ID=%ld] is attached\n",name,id);
}
```

4.1.1.2. Retrieving Interface Information

INFO_GET_CHANNELS

Return the number of channels a module has.

INFO_GET_VECTOR_SIZE

Get the number of bytes needed to store the I/O values. This is the size you pass with the VECIN/VECOUT calls.

INFO_GET_PORT_CLASS

Returns the class information as a 16-Bit bitmap. The values that can be returned are:

CLASS_DIGITAL, CLASS_ANALOG, CLASS_PWM, CLASS_FREQ, CLASS_EVTCNT, CLASS_POS, CLASS_INTEGER, CLASS_FLOAT.

Any of those values can occur in any combination with CLASS_INPUT or CLASS_RMW. If both is not set, the device is of class CLASS_OUTPUT.

4.1.1.3. Retrieving Hardware State Information

INFO_GET_POWERFAIL

Returns TRUE if a powerfail was detected.

INFO_GET_WATCHDOG

Returns TRUE if the on-board watchdog was not triggered in time and the outputs where cleared.

4.1.1.4. Enable Operation

CFG_ENABLE

Some devices need to be enabled before use. This is true for analog output devices that have been designed for motion control applications. Those devices have sometimes watchdog relays to cut the analog output and to pull the terminal down to 0 to stop an attached servo controller. In this type of applications, its necessary to enable the device before use. This is done by calling CFG or CFGCH with *CFG_ENABLE* as a cmd and TRUE as a *value*.

Enable is also used with devices that share hardware. Example: The TSMARMCPU and the TSMCPU32H2 use hardware counters to provide a mix of event and frequency counters. You can have up to 2 event counters or up to 8 frequency counters or 1 event and 4 frequency counters (that is because you can easily multiplex a frequency counter but not a event counter). The configuration used is just determined by a *CFG_ENABLE.* In this case, the *value* passed is 0 to disable the frequency counters and non-null to set the counters gate time. Please refer to for I.4.3.1.1. TSM-ARMCPU details.

4.1.2. Analog I/O

4.1.2.1. Preferred Physical Units

ExpressIO^{TM} uses integer arithmetic for several reasons. Thus it is not always possible to return the value in a physical base unit. As an example, we return temperatures multiplied by 10.

What to measure	***Scale factor***	***Physical Unit***
Temperatur	d (dezi) 10^{-1}	Degrees (°C)
Voltage	m (milli) 10^{-3}	Volt
Current	µ (micro) 10^{-6}	Ampere

4.1.2.2. CFG & INFO Calls Special to Analog Modules

CFG_SET_CHANNEL_RANGE

The value passed is a range identifier. Not all ranges can be used with all analog inputs. The TSM-8AD12 is the only board where you can set the electrical input range on a channel-by-channel base. All others have a fixed input range (0..5V) or device specific input ranges (TSM-8AD8<KTY and TSM-8AD8<TCK).

Range identifier	***Physical input range / scaled return value***	***Analog input module***
RANGE_RAW	Binary	ALL
RANGE_RAW_U_10000	0..4095, 0..10V	TSM-8AD12 ONLY
RANGE_RAW_U_5000	0..4095, 0..5V	TSM-8AD12 ONLY
RANGE_RAW_S_10000	-2048..+2047, -10..10V	TSM-8AD12 ONLY
RANGE_RAW_S_5000	-2048..+2047, -5..5V	TSM-8AD12 ONLY
RANGE_U_10000	0..10000mV	TSM-8AD12
RANGE_U_5000	0..5000mV	ALL
RANGE_S_10000	-10000..10000mV	TSM-8AD12, TSM-2DA12
RANGE_S_5000	-5000..5000mV	TSM-8AD12

Range identifier	***Physical input range / scaled return value***	***Analog input module***
RANGE_UB[***]	0..1080dV (typ. 80dV..330dV)	DINX
RANGE_020mA	0..20000µA (250 Ohm shunt)	ALL[*]
RANGE_420mA	0..20000µA (250 Ohm shunt)	ALL[*]
RANGE_PT100V4	-500..+2050 d°C	ALL[*]
RANGE_KTY	-250..1020+ d°C	TSM-8AD8<KTY, DINX, TSMCPUH2
RANGE_LM34	-150..+2500 d°C	ALL 0..5000mV inputs, DINX
RANGE_THERMO_K	-250..+2300 d°C	TSM-8AD8<TCK

(*) Hardware modification maybe required (shunt resistor)

(**) External PER-PT100V4 module required

(***) RANGE_UB is used to for the DINX operating voltage sensor (typ. 8V..33V)

CFG_SET_GAIN

If you need to customize an analog input you can set an individual gain factor using this call. This value has to be a 16-Bit signed integer.

For more information see I.4.1.2.3. Setting individual scaling factors

CFG_SET_ATTENTUATE

If you need to customize an analog input you can set an individual attenuate factor using this call. This value has to be a 16-Bit signed integer.

For more information see I.4.1.2.3. Setting individual scaling factors

CFG_SET_OFFSET

If you need to customize an analog input you can set an individual offset using this call. This value has to be a 32-Bit signed integer.

For more information see I.4.1.2.3. Setting individual scaling factors

CFG_SET_LINTAB

If you need to customize an analog input you can attach a LINTAB linearisation table. The table is then used to correct and to scale all inputs.

For more information see I.4.1.2.3. Setting individual scaling factors

INFO_GET_CHANNEL_RANGE

Returns the range identifier set using CFG_SET_CHANNEL_RANGE.

4.1.2.3. Setting Individual Scaling Factors

The conversion of the analog values is done by a linear conversion algorithm:

$$retval = \frac{gain}{attentuate} * rawvalue + offset$$

Because we use integer arithmetic only, the actual form is a bit different:

$$retval = \frac{gain * rawvalue}{attentuate} + offset$$

Executing the multiplication first gives a better precision of the result.

It is possible to create a private scaling using the standard calls CFG_SET_GAIN, CFG_SET_ATTENTUATE and CFG_SET_OFFSET. First, set the desired physical input range like ±5V (RANGE_RAW_U_5000), if the hardware supports different ranges. And then use the above calls to set the individual scaling rules:

Presume we are using a TSM-8AD12 module:

We want to use a pressure sensor with linear output. The physical range shall be 2.1..8.6V, which fits into the input range of 0..10V. The pressure range is 40000-153000Pa (pascal).

A linear curve follows the formula:

p = delta * U + offset

where p is the pressure, delta is the gain and U the input voltage.

We can calculate delta from the values we have:

delta = (y2-y1) / (x2-x1)

= (153000 - 40000) / (8.6 - 2.1)

= 17384.6 Pa/V

We have delta now, so we can calculate the offset:

offset = p - delta * U

= 40000Pa - 2.1V * 17384.6 Pa/V

= 3492Pa

Finally we have to multiply delta by the conversion factor of the AD converter. For a 12-Bit ADC (TSM-8AD12) and an **RAW** input range of 0..10V DC it is:

step = 10V / 4096 digits

= 0.002441 V/digit

$\text{delta}_{\text{dig}}$ = delta * steps

= 17384.6 Pa/V * 0.002441 V/digit

= 42.44 Pa/digit

As ExpressIO$^{\text{TM}}$ uses integer arithmetic, we have to split the floating point value »42.44« into two integer values. Be aware that both values are signed and that they have to be in the range -32767...+32768.

gain = 4244, attenuation = 100

gain / attentuation = $\text{delta}_{\text{dig}}$

```
IOObjCreate(&pressure,NULL,BUS_TYPE_TSM,8,0,CLASS_ANALOG,NULL);
CFG(pressure, CFG_SET_CHANNEL_RANGE, RANGE_RAW_U_10000);
CFG(pressure, CFG_SET_GAIN, 4244);
CFG(pressure, CFG_SET_ATTENTUATE, 100);
CFG(pressure, CFG_SET_OFFSET, 3492);
```

See sample "SCAD.C" in express.io\demos directory.

> *All scaling factors set before are ignored when a LINTAB is attached and a LINTAB is dropped if scaling factors are set.*

4.1.2.4. Attaching an Individual Interpolation Table

Using the command line based program LINTAB.EXE you can create interpolation tables that can be used by the analog input drivers to linearize the input value. Because of the power and flexibility of LINTAB tables, you can include all physical parameters like gain and offset within such a table. LINTAB includes support for popular sensors like PT100 (RTD 100Ohm) and the thermocouples of type R, S, B, J, T, E, K.

The table consists of raw values expected by the analog inputs. A binary search algorithm is used to find the closest possible match and then a linear interpolation between the value and the next higher value is calculated. This simple algorithm is pretty fast and very precise. It has proved successful in analog measurement applications.

For detailed information on LINTAB, see the .

```
PRIVATE IOOBJECT temperature;

extern LINCTRL pt100v4;    // LINTAB generates a seperate C file. The name of the
                           // exported LINCTRL structure can be set as a LINTAB
                           // command line option

SYSTEM () {
   /* Create the object */
   IOObjCreate(&temperature,NULL,BUS_TYPE_TSM,13,0,CLASS_ANALOG,NULL);
   CFG(&temperature,CFG_SET_LINTAB,&pt100v4,);
}
```

All scaling factors set before are ignored when a LINTAB is attached and a LINTAB is dropped if scaling factors are set.

4.1.3. CFG & INFO Calls for Position Encoder Drivers

CFG_POS_SET_DIR

If set to 1 the counter direction is inverted.

CFG_SSI_SET_TURNS, CFG_SSI_SET_STEPS

Absolute Position encoders with *SSI*-Interface specifies it's resolution in *TURNS* and *STEPS*. CFG_SSI_SET_TURNS sets the SSI *TURNS* parameter. Usual values are 512/1024. TURNS are the numbers of revolutions the encoder can resolve. CFG_SSI_SET_STEPS is used to set the number of turns. From the driver you get an absolute position that is composed of STEPS and TURNS to form an integral position value.

*Please note that both values are in **bits***!

Example:

TURNS	512	= 9 bits
STEPS	1024	= 10 bits

```
CFG(ssi,CFG_SSI_SET_TURNS,9);

CFG(ssi,CFG_SSI_SET_STEPS,10);
```

INFO_GET_INDEX_STATUS

The value reported by the driver is 9+10 = 19 bits [0..524287]. If there is an over- or underflow of the boundary values, the encoder will jump to the opposite value:

underflow:	0-> 524287
overflow:	524287->0

4.1.4. XP BASIC Info and Configuration CALLS

4.1.4.1. Retrieving the Name of an ExpressProgram

XP_INFO_GET_NAME

Returns the pointer to the name of the XP. Useful for debugging purposes.

param: Not used (0)

return: pointer to an ASCIIZ string

example:

```
    char xpname; /* we assume an IOOBJECT »my_ioobject is declared */
    xpname = (char *) INFO(&my_ioobject, XP_INFO_GET_NAME, 0);
    if (xpname) {
        sprintf(string,"Express Program '%s' is attached\n",xpname);
    }
```

4.1.4.2. Setting and Retrieving the Sample Rate

XP_INFO_GET_SAMPLE_RATE

Returns the currently used sample rate of an XP in ms.

param: Not used (0)

return: The sample rate in milliseconds

example:

```
    INT32 smpl; /* we assume an IOOBJECT »my_ioobject is declared */
    smpl = (INT32) INFO(&my_ioobject, XP_INFO_GET_SAMPLE_RATE, 0);
    sprintf(string,"Used sample rate is '%ld',\n", smpl);
```

XP_INFO_GET_MAX_SAMPLE_RATE

Returns the maximal possible sample rate. This value is limited by the used system and some possible requirements of an XP (small value = high sample rate!). With TSM900, the maximum rate is 5ms.

param: Not used (0)

return: The maximal sample rate in milliseconds

example:

```
    INT32 smpl; /* we assume an IOOBJECT »my_ioobject is declared */
```

```
smpl = INFO(&my_ioobject, XP_INFO_GET_MAX_SAMPLE_RATE, 0);

sprintf(string,"Maximum sample rate is '%ld',\n", smpl);
```

XP_CFG_SET_SAMPLE_RATE

Set the sample rate an XP should work with. The sample rate cannot be smaller than the minimum sample rate (with TSM900, the maximum rate is 5ms).

PARAM : Sample rate in milliseconds

```
/* set » my_ioobject« to 50 milliseconds sample rate */

CFG(&my_ioobject, XP_CFG_SET_SAMPLE_RATE, 50);
```

4.2. ExpressPrograms for DIGITAL I/O

By default, there are 5 XP's available, all extending the DIGITAL input and output drivers. However, XP's can be added to an already running system. If you have needs for a special XP, please contact mocom software (support@mocom-software.de).

4.2.1. Single Channel XP's

These XP's support a single channel (IN(), OUT()) only.

4.2.1.1. EDGE DETECTOR

This ExpressProgram monitors a given i/o port to detect level changes. The user can pass a trigger condition to the EDGE device by calling a WAIT, WAITIO or a XPEventSubscribe call. When the condition is matched, all pending WAIT and WAITIO calls waiting for a specific EDGE device are resumed. All pending XPEvent messages are send back to the application they where posted by.

Property	***Comment***
Name	EDGE
Max. Detection Frequency	100Hz
Source of i/o Events?	*YES*
Trigger	IO_EVT_FALL, IO_EVT_RISE, IO_EVT_BOTH, IO_EVT_ONE,IO_EVT_ZERO
Special CFG calls	-
Special INFO calls	-

Table 2: EDGE XP Quick Overview

4.2.1.2. EVENT COUNTER

The event counter is a 32-Bit counter that can be attached to any digital input channel. The maximum input frequency is 100Hz (symmetric, 50% duty cycle). COUNTER modifies the functions IN and OUT. With IN() you can read the current counter value and with OUT() you can override the counter with an value - like »0« to reset it.

Property	***Comment***
Name	COUNTER
Max. Detection Frequency	100Hz
Source of i/o Events?	*NO*
Trigger	-
Special CFG calls	-
Special INFO calls	-

Table 3: COUNTER XP Quick Overview

4.2.1.3. PULSE

The PULSE device is designed to generate:

- MONOFLOP like timed pulses
- Delayed ON
- Delayed pulses
- Pulse sequences with variable duty cycle

Property	***Comment***
Name	PULSE
Max. Detection Frequency	100Hz
Source of i/o Events?	*NO*
Trigger	-
Special CFG calls	XP_CFG_SET_LOWTIME,XP_CFG_SET_HIGHTIME, XP_CFG_SET_TRIGGERMODE,XP_CFG_SET_INVERSION
Special INFO calls	XP_INFO_GET_LOWTIME,XP_INFO_GET_HIGHTIME, XP_INFO_GET_TRIGGERMODE,XP_INFO_GET_INVERSION

Table 4: PULSE XP Quick Overview

The PULSE-XP can be assigned to all digital outputs. It is configurable using the CFG() macro. It is triggered by an OUT(<*pulse-device*>,<*value*>) call. <*value*> is added to a device internal counter called *queue counter*. The device can be reset to its initial state using OUT(<*pulse-device*>,0) at any time (*queue counter is cleared, too*). If trigger mode is TRG_QUEUED the *queue counter* is decreased and the next pulse is started after each pulse until the *queue counter* is 0. So its possible to output a well defined sequence of pulses.

To understand the PULSE device, you have to know the four states a pulse device can be in:

1. IDLE STATE

An unused PULSE device is in the IDLE state. This means: “do nothing”.

2. SYNC STATE

When a PULSE device is triggered it shifts to the SYNC state. In SYNC state it waits for the next clock tick to occur. If the tick arrieves, it sets the assigned output to the level defined for the LOW STATE ('0' if normal, '1' if inverted).

3. LOW STATE

The Device counts down the low time by one with every tick until it reaches '0'. Then it sets the assigned output to the level defined for the HIGH STATE ('1' if normal, '0' if inverted). If the high time is zero, the device falls back to the IDLE STATE.

4. HIGH STATE

The Device counts down the high time by one with every tick until it reaches '0'. Then it sets the assigned output to the level defined for the LOW STATE ('0' if normal, '1' if inverted). If the trigger mode is not TRG_QUEUED or the queue counter is zero, it falls back to the IDLE STATE afterwards. If trigger mode is TRG_QUEUED and the queue counter is not '0', the counter is decremented and a new cycle begins.

To create a MONOFLOP, configure the PULSE device as follows:

- INVERTED = TRUE
- LOWTIME: The time you like the output to be active (high)
- HIGHTIME = 0
- TRIGGER: TRG_SINGLE or TRG_RETRIGGER

To create a MONOFLOP with defined low time before the PULSE, configure the PULSE device as follows:

- INVERTED = FALSE
- LOWTIME: The desired delay befor pulse
- HIGHTIME: The time you like the output to be active (high)
- TRIGGER: TRG_SINGLE or TRG_RETRIGGER

To create a MONOFLOP with defined low time after the PULSE, configure the PULSE device as follows:

- INVERTED = TRUE
- LOWTIME: The active high time
- HIGHTIME: The delay after the pulse
- TRIGGER: TRG_SINGLE or TRG_RETRIGGER

To create a delayed ON Function:

- INVERTED = FALSE
- LOWTIME: The desired delay
- HIGHTIME = 0
- TRIGGER: TRG_SINGLE or TRG_RETRIGGER

To create a PULSE SEQUENCE GENERATOR:

- INVERTED = FALSE
- LOWTIME: The PULSE LOWTIME
- HIGHTIME: The PULSE HIGHTIME
- TRIGGER: TRG_QUEUED

KEYWORD	***Comment***	***MACRO***
XP_CFG_SET_LOWTIME	Set the OFF period in milliseconds (deadtime), can be 0.	CFG
XP_INFO_GET_LOWTIME	Retrieve the current off time	INFO
XP_CFG_SET_HIGHTIME	Set the ON period in milliseconds (pulse)	CFG
XP_INFO_GET_HIGHTIME	Retrieve the current pulse width	INFO
XP_CFG_SET_TRIGGERMODE	Set trigger mode. The argument can be: TRG_SINGLE TRG_RETRIGGER TRG_QUEUED	CFG
XP_INFO_GET_TRIGGERMODE	Retrieve the current trigger mode	INFO
XP_CFG_SET_INVERSION	Set inversion mode. If TRUE, the logical output is inverted (LOWTIME is “ON”, HIGHTIME is “OFF”).	CFG
XP_INFO_GET_INVERSION	Retrieve the current invert state	INFO

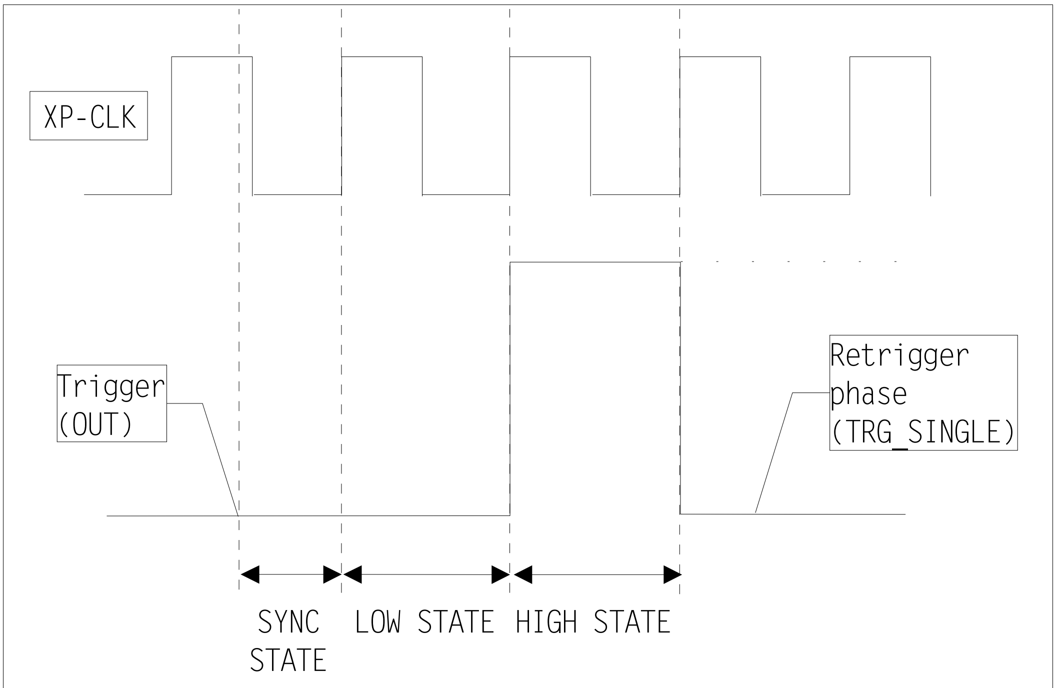


Figure 17: Pulse device: functional description

The Trigger-Modes differ and allow a wide variety of applications:

Mode	***Description***	***Application***
TRG_SINGLE	A new trigger is accepted in the IDLE STATE only – that is after the current pulse was completed.	non-retriggerable monoflops
TRG_RETRIGGER	Same as TRG_SINGLE but the trigger is also accepted in the LOW STATE to extend the LOW STATE. The HIGH STATE is not influenced by this trigger.	retriggerable monoflops
TRG_QUEUED	All trigger pulses are counted. At the end of a pulse the trigger counter is decremented and as long as the counter is not 0, a new pulse is started.	Pulse sequence generator

Whenever the output macro is called with value other than 0, the queue counter is incremented by the given value and device is triggered.

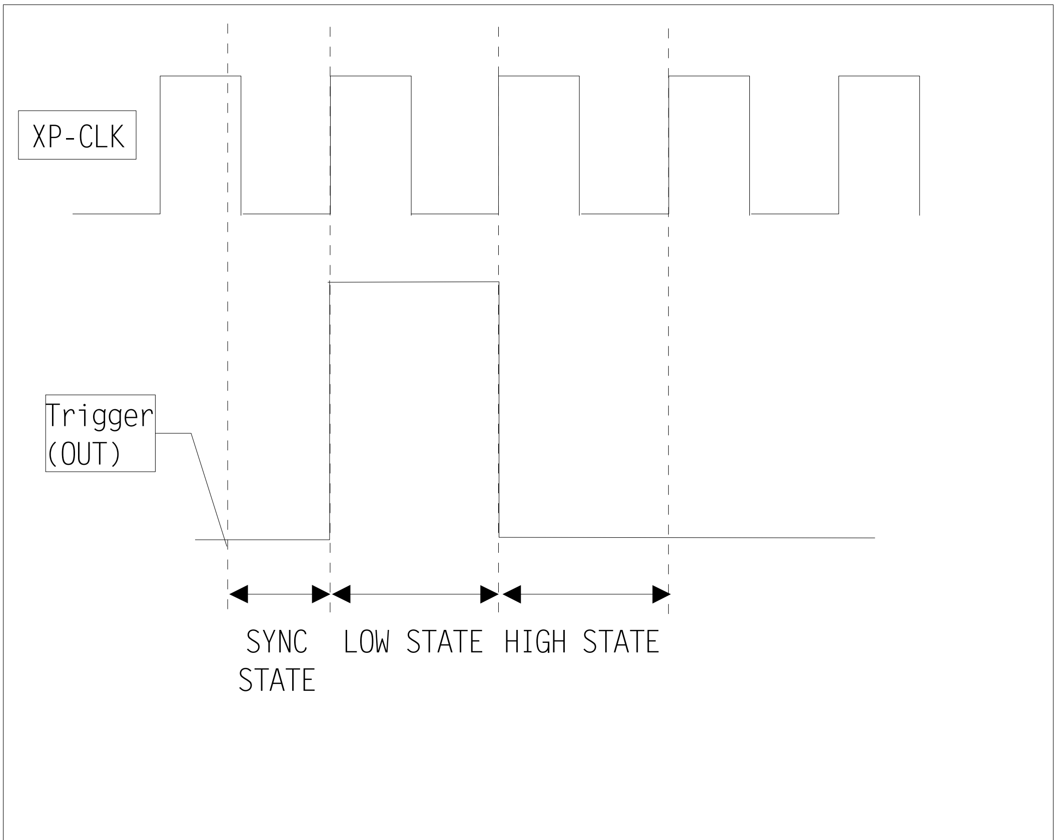


Figure 18: Pulse device: functional description - inverted

Example:

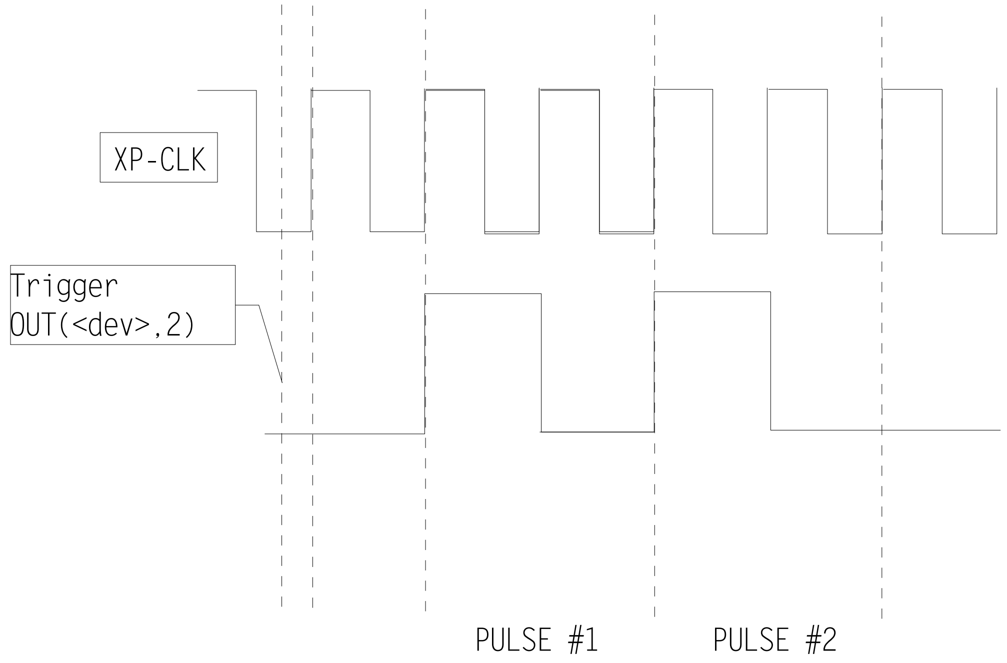


Figure 19: Pulse sequence generator

```
SYSTEM ()
{
    // create ioobject with PULSE XP
    IOObjCreate(&pulse,         /* the device                           */
                NULL,           /* no name                              */
                BUS_TYPE_CPU,   /* it is located on the CPU BUS         */
                CPU_DOUT,       /* module address                       */
                0,              /* CHANNEL                              */
                CLASS_DIGITAL,  /* it is a DIGITAL output module        */
                "PULSE");       /* USE PULSE XP                         */

    // configure pulse device: We want a MONOFLOP with 100ms ON-TIME
    // there is no need to set the configuration in a specific sequence
    CFG(&pulse,XP_CFG_SET_INVERTION,TRUE);            // INVERSION
    CFG(&pulse,XP_CFG_SET_TRIGGERMODE,TRG_SINGLE);    // NO RETRIGGER
    CFG(&pulse,XP_CFG_SET_HIGHTIME,0);                // HIGHTIME = 0
    CFG(&pulse,XP_CFG_SET_LOWTIME,100);               // LOWTIME = 100 ms
}
TaskMain ()
{
    SYSTEM();
    ...
    OUT(&pulse,1);        // trigger MONOFLOP
    ...
}
```

4.2.2. Vector XP's

Vector XP's operate on several channels at a time. The channels must be located on the same module. If an IOOBJECT is created for use with a vector XP it automatically inherits the number of channels available from the underlying physical device. For instance, with a TSM-8E24 »channel« should be set to 8 to form 8 counters.

Note: In earlier versions of ExpressIO^{TM} the ***channel*** *parameter was used to set the number of channels available. This is no longer supported. However, a change of existing software should not be needed.*

The Vector XP's are VERY powerful and efficient! They consume a minimum of processor power. However, you will find it more flexible to use single channel XP's in many applications.

4.2.2.1. VECCOUNTER

This XP is very similar to the single channel "COUNTER" device. In contrast to »COUNTER«, this device is capable of making a TSM32E24 a 32 channel 32-bit event counter for frequencies up to 100Hz with only minimum overhead! The counters can be read and written using VECIN and VECOUT calls. Using VECOUT/VECIN will guarantee that all counters are read or written at a time - without being interrupted by the counting process.

Property	***Comment***
Name	VECCOUNTER
Max. Detection Frequency	100Hz
Source of i/o Events?	*NO*
Trigger	**-**
Special CFG calls	-
Special INFO calls	-

Table 5: VECCOUNTER XP Quick Overview

4.2.2.2. CAPTURE

While VECCOUNTER is similar to COUNTER, CAPTURE is similar to EDGE. However, the differences are bigger:

CAPTURE will scan a digital input device (like the TSM8E24) for level changes. If an edge is detected and a thread is waiting for this XP, the thread is triggered and the current IO state is saved within the XP. You can use IN() and VECIN() macros to read these captured values. However, it is by far more efficient to use WAITIO or to pass a data buffer by XPEventSubscribe call to read the captured data. See the »CAPTURE.C« sample.

A good trick is to use two IOObjects: One without an XP and one with CAPTURE attached at the same time. So you can access the captured and the raw value as you need.

Property	***Comment***
Name	CAPTURE
Max. Detection Frequency	100Hz
Source of i/o Events?	*YES*
Trigger	IO_EVT_FALL, IO_EVT_RISE, IO_EVT_BOTH, IO_EVT_ONE,IO_EVT_ZERO
Special CFG calls	-
Special INFO calls	-

Table 6: CAPTURE XP Quick Overview

4.3. Supported Hardware Reference

4.3.1. ELZET80 TSM

4.3.1.1. TSM-ARMCPU

The TSMARMCPU on-board I/O can be accessed by using the BUS_TYPE_CPU Bus selector and the module and channel addresses found below:

Module	***Channel***	***Range***	***Associated PINs***	***Comment***
CPU_AIN	0..7	0..5V, 4..20mA, KTY, PT100V4 or LM34. 12Bit	AIN I1..8	Can be used with PT100V4*
CPU_AOUT	0,1	0..10V / 8Bit	AOUT O1,2	
CPU_DIN	0..7	24VDC	DIN I1..8	Can be used by frequency counters
CPU_DOUT	0..7	24VDC / 0.5A	DOUT O1..8	
	8	RELAY	REL	

Module	***Channel***	***Range***	***Associated PINs***	***Comment***
CPU_FREQ**	0..7	24VDC	DIN I1..8	24-Bit HW counter up to 500Hz
CPU_EVTCNT**	0,1	24VDC	DIN I1,4	24-Bit HW counter up to 500Hz

* External PER-PT100V4 module required

** You can't use the frequency counter and the hardware event counter at the same time.

If you connect an LM34 Temperature sensor to a 0..5000mV input, you can set RANGE_LM34 to convert the voltage to 1/10 °C (d°C).

The default configuration is:

Module/Channel	***Default Range***	***Associated PINs***	***Switch SW600***	
			0..20mA	***KTY***
AIN 0	0..5000 [mV]	AIN I1	1:OFF	2:OFF
AIN 1	0..5000 [mV]	AIN I2	3:OFF	4:OFF
AIN 2	0..5000 [mV]	AIN I3	5:OFF	6:OFF
AIN 3	0..5000 [mV]	AIN I4	7:OFF	8:OFF
			Switch SW601	
AIN 4	0..5000 [mV]	AIN I5	1:OFF	2:OFF
AIN 5	0..5000 [mV]	AIN I6	3:OFF	4:OFF
AIN 6	0..5000 [mV]	AIN I7	5:OFF	6:OFF
AIN 7	0..5000 [mV]	AIN I8	7:OFF	8:OFF
AOUT 0	0..10000 [mV]	AOUT O1		
AOUT 1	0..10000 [mV]	AOUT O2		

The analog inputs can be configured by use of the switches SW600/601 to accept different sensors:

SW600	***0..5000mV***		***0..20mA***		***KTY***	
	RANGE_U_5000		RANGE_020mA		RANGE_KTY	
AIN 0	1:OFF	2:OFF	1:ON	2:OFF	1:OFF	2:ON
AIN 1	3:OFF	4:OFF	3:ON	4:OFF	3:OFF	4:ON
AIN 2	5:OFF	6:OFF	5:ON	6:OFF	5:OFF	6:ON
AIN 3	1:OFF	2:OFF	1:ON	2:OFF	1:OFF	2:ON
SW601	***0..5000mV***		***0..20mA***		***KTY***	
AIN 0	1:OFF	2:OFF	1:ON	2:OFF	1:OFF	2:ON

SW600	***0..5000mV***		***0..20mA***		***KTY***	
AIN 1	3:OFF	4:OFF	3:ON	4:OFF	3:OFF	4:ON
AIN 2	5:OFF	6:OFF	5:ON	6:OFF	5:OFF	6:ON
AIN 3	7:OFF	8:OFF	7:ON	8:OFF	7:OFF	8:ON

Please note that TSMARMCPU can not automatically adapt the input conversion – the switches cannot be read back. It is the responsibility of the Programmer to set-up the ranges as needed.

4.3.1.1.1. The Hardware Frequency and Event counters

The TSMARMCPU uses two hardware counters to implement frequency measurement and event counters. The major difference between event counting and frequency measurement is that event counting occupies a hardware counter while with frequency measurement you can use a multiplexer to count serveral channels with a single hardware counter. The TSMARMCPU uses two 4-to-1 multiplexers to support up to 8 channels for frequency counting. See figure Figure 18 (Page 187) for the counter architecture of TSMARMCPU.

Possible Combinations:

Number of FREQ/EVENT channels	***Frequency measurement***		***Event counter***	
	channels	***inputs***	***channel(s)***	***Input(s)***
4f / 1e	0..3	1..4	1	5
1e / 4f	4..7	5..8	0	1
2e	*none*	*none*	0..1	1,5
8f	0..7	1..8	*none*	*none*

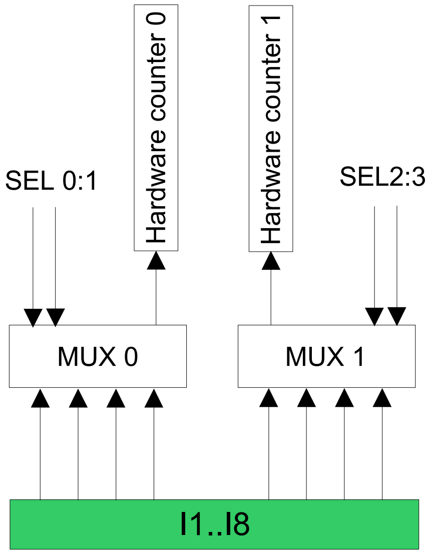


Figure 20: Counter and Multiplexer architecture of TSMCPUH2

CFG_SET_ENABLE

Enable frequency counting on a given channel. The parameter you pass with CFG_SET_ENABLE is the gate time for the counter. You can enable 4 frequency counters at a time (see Figure 18 on page 187). For each group of four frequency counters an individual gate time can be set.

input freq range	***Recommended GATE time***	***output***
> 10000 Hz	100ms	HZ
< 10000 Hz	1000ms	HZ
< 100 Hz	10000ms	HZ

Please note that frequencies higher than 3000Hz can not be measured without a modification of the hardware.

4.3.1.2. TSM-CPUH2

The TSMCPUH2 on-board I/O can be accessed by using the BUS_TYPE_CPU Bus selector and the module and channel addresses found below:

Module	***Channel***	***Range***	***Associated PINs***	***Comment***
CPU_AIN	0..7	0..5V, 4..20mA, KTY, PT100V4 or LM34. 10Bit	AIN I1..8	Can be used with PT100V4[*]
CPU_AOUT	0,1	0..10V / 8Bit	AOUT O1,2	
CPU_DIN	0..7	24VDC	DIN I1..8	Can be used by frequency counters
CPU_DOUT	0..7	24VDC / 0.5A	DOUT O1..8	
	8	RELAY	REL	
CPU_PWM	0,1	0..1000‰	DOUT O1,O2	8-Bit, 600HZ PWM
CPU_FREQ[**]	0..7	24VDC	DIN I1..8	32-Bit HW counter up to 500Hz
CPU_EVTCNT[**]	0,1	24VDC	DIN I1,4	32-Bit HW counter up to 500Hz

[*] External PER-PT100V4 module required

[**] You can't use the frequency counter and the hardware event counter at the same time.

If you connect an LM34 Temperature sensor to a 0..5000mV input, you can set RANGE_LM34 to convert the voltage to 1/10 °C (d°C).

The default configuration is:

Module/Channel	***Default Range***	***Associated PINs***	***Switch SW600***	
			0..20mA	***KTY***
AIN 0	0..5000 [mV]	AIN I1	1:OFF	2:OFF
AIN 1	0..5000 [mV]	AIN I2	3:OFF	4:OFF
AIN 2	0..5000 [mV]	AIN I3	5:OFF	6:OFF
AIN 3	0..5000 [mV]	AIN I4	7:OFF	8:OFF

Module/Channel	***Default Range***	***Associated PINs***	***Switch SW600***	
			0..20mA	***KTY***
			Switch SW601	
AIN 4	0..5000 [mV]	AIN I5	1:OFF	2:OFF
AIN 5	0..5000 [mV]	AIN I6	3:OFF	4:OFF
AIN 6	0..5000 [mV]	AIN I7	5:OFF	6:OFF
AIN 7	0..5000 [mV]	AIN I8	7:OFF	8:OFF
AOUT 0	0..10000 [mV]	AOUT O1		
AOUT 1	0..10000 [mV]	AOUT O2		

The analog inputs can be configured by use of the switches SW600/601 to accept different sensors:

SW600	***0..5000mV***		***0..20mA***		***KTY***	
	RANGE_U_5000		RANGE_020mA		RANGE_KTY	
AIN 0	1:OFF	2:OFF	1:ON	2:OFF	1:OFF	2:ON
AIN 1	3:OFF	4:OFF	3:ON	4:OFF	3:OFF	4:ON
AIN 2	5:OFF	6:OFF	5:ON	6:OFF	5:OFF	6:ON
AIN 3	1:OFF	2:OFF	1:ON	2:OFF	1:OFF	2:ON
SW601	***0..5000mV***		***0..20mA***		***KTY***	
AIN 0	1:OFF	2:OFF	1:ON	2:OFF	1:OFF	2:ON
AIN 1	3:OFF	4:OFF	3:ON	4:OFF	3:OFF	4:ON
AIN 2	5:OFF	6:OFF	5:ON	6:OFF	5:OFF	6:ON
AIN 3	7:OFF	8:OFF	7:ON	8:OFF	7:OFF	8:ON

Please note that TSMCPUH2 can not automatically adapt the input conversion – the switches cannot be read back. It is the responsibility of the Programmer to set-up the ranges as needed.

4.3.1.2.1. The Hardware Frequency and Event counters

The TSMCPUH2 uses two hardware counters to implement frequency measurement and event counters. The major difference between event counting and frequency measurement is that event counting occupies a hardware counter while with frequency measurement you can use a multiplexer to count serveral channels with a single hardware counter. The TSM-CPUH2 uses two 4-to-1 multiplexers to support up to 8 channels for frequency counting. See figure Figure 17 (Page 186) for the counter architecture of TSMCPUH2.

Possible Combinations:

Number of FREQ/EVENT channels	***Frequency measurement***		***Event counter***	
	channels	***inputs***	***channel(s)***	***Input(s)***
4f / 1e	0..3	1..4	1	5
1e / 4f	4..7	5..8	0	1
2e	*none*	*none*	0..1	1,5
8f	0..7	1..8	*none*	*none*

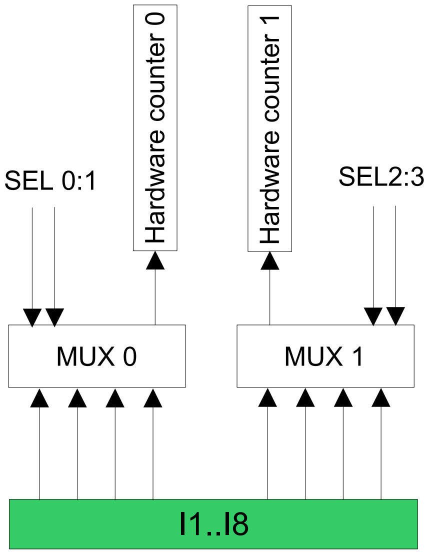


Figure 21: Counter and Multiplexer architecture of TSMCPUH2

CFG_SET_ENABLE

Enable frequency counting on a given channel. The parameter you pass with CFG_SET_ENABLE is the gate time for the counter. You can enable 4 frequency counters at a time (see Figure 17 on page 186). For each group of four frequency counters an individual gate time can be set.

input freq range	***Recommended GATE time***	***output***
> 10000 Hz	100ms	HZ
< 10000 Hz	1000ms	HZ
< 100 Hz	10000ms	HZ

Please note that frequencies higher than 3000Hz can not be measured without a modification of the hardware.

4.3.1.3. TSM-CPU900

The CPU900 on-board I/O can be accessed by using the BUS_TYPE_CPU Bus selector and the module and channel addresses found below:

Module	***Channel***	***Physical Range***	***Associat-ed PINs***	***Comment***
CPU_AIN	0..3	0..5V, 10Bit	A1..4	Can be used with PT100V4** or as 0..20mA with 250 Ohm shunt*
CPU_DOUT	0,1	Relay	R1,R2	
CPU_DOUT	2,3	24VDC / 2A	O1,O2	Can be used by PWM devices
CPU_PWM	0,1	0..1000‰	O1,O2	Physically same as DOUT chan 0
CPU_DIN	0..3	24VDC	I1..4	
CPU_EVTCNT	0,1	24VDC	I3,I4	16-Bit HW counter up to 500Hz

(*) Hardware modification required (shunt resistor)

(**) External PER-PT100V4 module required

4.3.1.4. Digital I/O boards

(8A24, 8E24, 16A24P, 16E24, 32A24, 32E24, 8A230, 8E230, 8REL)

This driver supports several TSM I/O modules, all with a common physical structure:

- Digital outputs and inputs
- Each bit in an IO-word is a physical I/O
- 8, 16 or 32 IO-channels per module
- All outputs can be read back
- Powerfail control logic
- Watchdog control logic (output modules only)
- All boards isolated

The following modules are supported by TSMDIO driver:

Name	***I/O***	***Number of channels***	***Voltage***	***Current***	***Technology***	***Comment***
TSM8A24	OUT	8	24VDC	0.5A	Transistors	
TSM8E24	IN	8	24VDC	-	-	
TSM16A24P	OUT	16	24VDC	1.5A	Transistors	
TSM16E24	IN	16	24VDC	-	-	
TSM32A24	OUT	32	24VDC	0.5A	Transistors	Total current limit = 5A
TSM32E24	IN	32	24VDC	-	-	
TSM8A230	OUT	8	230VAC	8A	Relay	Total current limit = 8A
TSM8E230	IN	8	230VAC	-	-	
TSM8REL	OUT	8	60V	2A	Relay	Changeover

4.3.1.5. Analog I/O modules

4.3.1.5.1. TSM8AD8

TSM8AD8 is a low cost analog to digital conversion module. It offers a limited resolution (8-Bits) and conversion speed (>150µs) at a really low price. This board is designed for applications like temperature measurement. There are 4 variants of that board available:

NAME	***SIGNALS***	***RANGE***	***RETURN (IN,VECIN)***	***COMMENT***
TSM8AD8	Voltage	0..5V	0..5000mV	

NAME	***SIGNALS***	***RANGE***	***RETURN (IN,VECIN)***	***COMMENT***
TSM8AD8	Voltage	-50..+205°C	-500..+2050 d°C**	PER-PT100V4. Set RANGE_PT100V4
TSM8AD8<I420	Current	2..21.2mA	200..21200 µA*	Common analog sensor interface
TSM8AD8<KTY	KTY Sensors	-25..+102°C	-250..+1020 d°C	KTY81, KTY10
TSM8AD8<TCK	Thermo-couples	-25..+230°C	-250..+2300 d°C	Type K Thermocouples

(*) Hardware modification required (shunt resistor)

(**) External PER-PT100V4 module required

The board is best for low cost temperature measurement using KTY81 sensors or the PT100V4 signal conditioner device for use with PT100 sensors. For use with PT100V4 the TSM8AD8 base module (0..5V) is used. Please keep in mind that you can connect the PT100V4 to all TSM analog input channels including the TSMCPU900 on-board channels.

The channels are really slow - it takes more than 150µs to mux, convert and read a channel. To improve performance, the channels are scanned in backgroud by an XP. If you access the channels using IN() or VECIN(), you will get a copy of a memory location where the most recently measured values are stored. A complete scan of all channels takes 10*8*2 = 160ms - for temperature applications this should never be critical ...

4.3.1.5.2. TSM8AD12

If the TSM8AD8 is not powerful enough, you may prefer the TSM8AD12. It supports fast conversion speed (~10µs per channel) and high resolution (12Bits). A special feature allows this module to re-configure the inputs to several ranges:

CFG Setting	***Input range***	***Raw Value***	***Return (IN,VECIN)***
RANGE_RAW_U_5000	0..5VDC	0..4095	0..4095
RANGE_RAW_U_10000	0..10VDC	0..4095	0..4095
RANGE_RAW_S_5000	-5..5VDC	-2048..+2047	-2048..+2047
RANGE_RAW_S_10000	-10..10VDC	-2048..+2047	-2048..+2047
RANGE_U_5000	0..5VDC	0..4095	0..5000mV
RANGE_U_10000	0..10VDC	0..4095	0..10000mV
RANGE_S_5000	-5..5VDC	-2048..+2047	-5000..+5000mV

CFG Setting	***Input range***	***Raw Value***	***Return (IN,VECIN)***
RANGE_S_10000	-10..10VDC	-2048..+2047	-10000..10000mV
RANGE_020mA*	0..20mA	0..4095	0..20000µA
RANGE_420mA*	4..20mA	0..4095	0..20000µA
RANGE_PT100V4**	-50..+205°C	0..4095	-500..+2050 d°C

(*) Hardware modification required (shunt resistor)

(**) External PER-PT100V4 module required

4.3.1.5.3. TSM2DA12

The TSM2DA12 has two channels with a resolution of 12 bits each. The output voltage is -10..+10VDC. Because this board was developed for motion control applications, the outputs are switched by relays to support secure reset operation and a fail safe watchdog action (0 VDC). The values to be set are in-between -10000..+10000mV, no other range is supported.

The outputs must be enabled using the CFG_SET_ENABLE configuration call. In releases of the “mCAT/TSM/2da12” driver before 1.10, the enable call will set/reset the internal enable marker only, to make the call effective you had to write a value to the DA-Converter. With the release 1.10 this changed. To disable a DA12 channel physically, you must call CFG_SET_ENABLE only, now. To enable it, nothing changed. First enable it, then output your values.

4.3.1.5.4. TSM4DA16

The TSM4DA16 is a 4 channel Digital-to-Analog module. The resolution is 16-Bit. The output voltage range is -10..+10VDC or 0..10VDC. The range can not be selected by software, the board needs a modification for that purpose. But the range can be detected by the software and the driver automatically is set up to work with the given range.

As with the TSM2DA12, a CFG_SET_ENABLE per channel is needed to enable the outputs.

4.3.1.6. Position Encoders

4.3.1.6.1. TSM4INC

TSM4INC is a 4 channel quadrature counter module. It supports the CFG_SET_DIR and INFO_GET_INDEX_STATUS calls, see I.4.1.2.5. Position encoder modules for details.

The counters are 24bit wide. When an index pulse was detected, it is stored in hardware until it is read by the INFO_GET_INDEX_STATUS call. To set the quadrature mode, use

CFG_SET_INC_MODE, INFO_GET_INC_MODE

There are 3 quadrature modes available:

LS7266_MODE_QUADRATURE_X1_COUNT

LS7266_MODE_QUADRATURE_X2_COUNT

LS7266_MODE_QUADRATURE_X4_COUNT

Default is LS7266_MODE_QUADRATURE_X4_COUNT

4.3.2. NET-A7

The NET-A7 base board itself has no ExpressIO aware I/O. However, there are a few extension boards available.

4.3.2.1. NET-A7-4I4R

The 4I4R has 4 digital inputs and 4 Relay driven digital outputs.

Module	***Channel***	***Range***	***Associated PINs***	***Comment***
CPU_DIN	0..3	24VDC	ST5,ST7	
CPU_DOUT	0..3	24VDC	ST6	Relays

4.3.3. EVA900

The EVA900 on-board I/O can be accessed by using the BUS_TYPE_CPU Bus selector and the module and channel addresses found below:

Module	***Channel***	***Range***	***Associated PINs***	***Comment***
CPU_AIN	0..3	0..5V, LM34, PT100V4*	ST70..ST73	
CPU_DIN	0..7	24VDC	ST30..ST31	
	8	TTL	ST7	DCF77 input
CPU_DOUT	0..3	24VDC / 0.5A**	ST52	
	4,5	RELAY	ST50,ST51	
	6	TRIAC	ST80	
CPU_PWM	0,1	24VDC / 0.5A**	ST52, PIN 4,5	Shared wit DOUT 1,2

Module	***Channel***	***Range***	***Associated PINs***	***Comment***
CPU_EVTCNT	0,1	24VDC	ST30, PIN 2,4	32-Bit HW counter up to 500Hz

[*] External PER-PT100V4 module required

[**] Total current for all outputs should never exceed 800mA.

If you connect an LM34 Temperature sensor to a 0..5000mV input, you can set RANGE_LM34 to convert the voltage to 1/10 °C (d°C).

If you want to use PT100 Temperature sensors, set RANGE_PT100V4 and connect an external ELZET80 “PERPT100V4” converter module to AIN I1..4.

The internal Timers are used to implement the counter and PWM features. Please refer to the following list for details on timer usage:

TLCS900 TIMER	***Use***	***Comment***
TIMER0	RC5 IR-Receiver	Used by RC5 driver
TIMER1	Not used	Use not recommended because of possible side effects
TIMER2	PWM	
TIMER3	mCAT/Ticker	
TIMER4	PWM	
TIMER5	Not used	Use not recommended because of possible side effects
TIMER6	Free for user	
TIMER7	Free for user	
TIMER8	Free for user	TI9 input of timer 8 is used for event counting using an HDMA. TI8 can be used to implement 16-Bit Counter/Timer applications

TLCS900 TIMER	***Use***	***Comment***
TIMER9	Free for user	TIB input of timer 9 is used for event counting using an HDMA. TIA can be used to implement 16-Bit Counter/Timer applications

4.3.4. DINX

All DINX control units have 8 24V DC inputs and 8 24V DC 0.5A outputs. The analog inputs vary depending on the DINX model:

Model	***CPU***	***Int. FLASH***	***Ext. FLASH***	***RAM***	***AIN***	***AOUT***
DINXF	95FY64	256	-	128kB	4 (0..5V)	-
DINXFA	95FY64	256	-	128kB	4 (0..5V 4..20mA KTY)	2

The DINX on-board I/O can be accessed by using the BUS_TYPE_CPU Bus selector and the module and channel addresses found below:

Module	***Channel***	***Range***	***Associated PINs***	***Comment***
CPU_AIN	0..7	0..5V, 4..20mA, KTY, PT100V4 or LM34. 10Bit	AIN 1..4	Channel 4 is connected to a internal LM34, Channel 5 measures the supply power.
CPU_AOUT	0,1	0..10V / 8Bit	AOUT 1,2	
CPU_DIN	0..7	24VDC	DIN 1..8	Can be used by event counters
CPU_DOUT	0..7	24VDC / 0.5A	DOUT 1..8	Can be used by PULSE ExpressPrograms
CPU_EVTCNT	0,1	24VDC	DIN 2,3	16-Bit HW counter up to 500Hz

(*) External PER-PT100V4 module required

If you connect an LM34 Temperature sensor to a 0..5000mV input, you can set RANGE_LM34 to convert the voltage to 1/10 °C (d°C).

If you want to use PT100 Temperature sensors, set RANGE_PT100V4 and connect an external ELZET80 “PERPT100V4” converter module to AIN I1..4.

The default configuration is:

Module/Channel	***Default Range***	***Associated PINs***	***Switch SW40***	
			0..20mA	***KTY***
AIN 0	0..5000 [mV]	AIN I1	1:OFF	2:OFF
AIN 1	0..5000 [mV]	AIN I2	3:OFF	4:OFF
AIN 2	0..5000 [mV]	AIN I3	5:OFF	6:OFF
AIN 3	0..5000 [mV]	AIN I4	7:OFF	8:OFF
AIN 4	-170..+330 [1/10 °C]	(internal sensor)		
AIN 5	1/10 V	(internal sensor)		
AIN 6	0..5000 [mV]	expansion connector		
AIN 7	0..5000 [mV]	expansion connector		
AOUT 0	0..10000 [mV]	AOUT O1		
AOUT 1	0..10000 [mV]	AOUT O2		

The analog inputs can be configured by use of switch SW40 to accept different sensors:

Module/Channel	***0..5000mV***		***0..20mA***		***KTY***	
RANGE	RANGE_U_5000		RANGE_020mA		RANGE_KTY	
AIN 0	1:OFF	2:OFF	1:ON	2:OFF	1:OFF	2:ON
AIN 1	3:OFF	4:OFF	3:ON	4:OFF	3:OFF	4:ON
AIN 2	5:OFF	6:OFF	5:ON	6:OFF	5:OFF	6:ON
AIN 3	7:OFF	8:OFF	7:ON	8:OFF	7:OFF	8:ON

Please note that DINX can not automatically adapt the input conversion – the switch cannot be read back. It is the responsibility of the Programmer to setup the ranges as needed.

4.3.5. I^{2}C-BUS

ExpressIO^{TM} supports up to 4 I^{2}C-8E8A and 4 I^{2}C-16E boards at a time. Those boards are designed to be I/O expanders for ELZET80 DINX.

4.3.5.1. Limitations

Currently the use of ExpressPrograms (Capture, Counter etc.) is prohibited with I^{2}C-Modules. The last Argument of IOObjCreate ***MUST*** be NULL.

4.3.5.2. I^{2}C-8E8A24

This module offers 8 digital inputs and 8 digital outputs. The outputs are protected by a watchdog timer. The timer will clear the outputs if the I/O is not updated at least every second. The output voltage is 24V, the maximum current per output is 0.5A.

In ExpressIO^{TM} a macro is used to calculate the “module” argument of an IOObjCreateCall:

I2C_8E8A24(m)

where (m) is the module address as selected by the address switch on the board. The valid range is 0..3.

From a software point of view, this module is a bit unique because it incorporates inputs and outputs in a single module:

Channel	***I/O***	***Access***
0..7	OUTPUTS	READ / WRITE
8..15	INPUTS	READ ONLY

4.3.5.3. I^{2}C-16E24

This module offers 16 24V inputs. They can be addressed as channels 0..15.

In ExpressIO^{TM} a macro is used to calculate the “module” argument of an IOObjCreateCall:

I2C_16E24(m)

where (m) is the module address as selected by the address switch on the board. The valid range is 0..3.

Example IOObjCreate call for a I^{2}C-Module

```
        /* create DIN on a PERDINX8E8A on address 4
         * the input channel is 1
         */
        IOObjCreate(&din8e8a24,     /* the device                             */
                    NULL,           /* no name                                */
                    BUS_TYPE_I2C,   /* it is located on the TSM BUS           */
                    I2C_8E8A24(4),  /* module address                         */
                    1,              /* CHANNEL                                */
                    CLASS_DIGITAL,  /* it is a DIGITAL input module           */
                    NULL            /* MUST BE NULL                           */
                   );
```

4.3.6. BITBAHN

The BITBAHN on-board I/O can be accessed by using the BUS_TYPE_CPU Bus selector and the module and channel addresses found below:

Module	***Channel***	***Range***	***Associated PINs***	***Comment***
CPU_AIN	0..3	0..10000mV / 10Bit	AIN I1..4	
CPU_AOUT	0..3	0..10000mV / 16Bit	AOUT O1..4	
CPU_DIN	0..7	24VDC	DIN I1..8	
CPU_DOUT	0..7	24VDC / 1.5A	DOUT O1..8	
CPU_EVTCNT	0,1	24VDC	DIN I1,2	32-Bit HW counter up to 500Hz

* External PER-PT100V4 module required

ExpressIO^{TM} support for BITBAHN is limited:

- No support for the 4 SSI inputs
- No support for -10..+10V mode of the digital to analog converters (DAC)
- No support for the 0..5V mode of the analog to digital converters (ADC)

5. Library Function Reference

The Library function interface is similar to the IOOBJECT interface. The major difference is that it allows direct access to physical drivers by use or bus, module and channel number. So these functions are used for special purposes, like the SYSMON express commands.

A basic limitation is that these functions can be used to access the physical driver only. No Access to IOOBEJCTS is possible (and so no access to ExpressPrograms is possible).

INTEGER IOCfg (UNSIGNED bus, UNSIGNED module, UNSIGNED chan, UNSIGNED cmd, UINT32 param);

Using IOCfg you can configure the driver. For allowed commands see the driver reference.

bus: ***Select the bus to be used.***

modules: ***Select a module within a bus.***

chan: ***Select a channel within the module (a single out-/input terminal)***

cmd: ***Is the command to be send to the driver***

param: ***A command specific argument to pass***

Return: ***TRUE specifies no error occurred***

UINT32 IOInfo (UNSIGNED bus, UNSIGNED module, UNSIGNED chan, UNSIGNED cmd, UINT32 param);

Using IOInfo you can read driver information. For allowed commands see the driver reference.

bus: ***Select the bus to be used.***

modules: ***Select a module within a bus.***

chan: ***Select a channel within the module (a single out-/input terminal)***

cmd: ***Is the command to be send to the driver***

param: ***A command specific argument to pass***

Return: ***The value returned by the driver. The type of this value can vary depending on the selected command.***

INT32 IOIn (UNSIGNED bus, UNSIGNED module, UNSIGNED chan);

Read from a single channel of a given module / bus. If the channel is a digital port, a "1" is returned if the port is active and a "0" if not.

bus: ***Select the bus to be used.***

modules: ***Select a module within a bus.***

chan: ***Select a channel within the module (a single out-/input terminal)***

void IOOut (UNSIGNED bus, UNSIGNED module, UNSIGNED chan, INT32 value);

Write to a single channel of a module / bus. If the channel is a digital output, it is activated if the parameter "value" is TRUE (not "0").

bus: ***Select the bus to be used.***

modules: ***Select a module within a bus.***

chan: ***Select a channel within the module (a single out-/input terminal)***

value: ***The value***

*INTEGER IOVecIn (UNSIGNED bus, UNSIGNED module, void *data, UNSIGNED len);*

Read all channels of a module with one call. »data« points to an array where the values are inserted. The size of an array item is depending on the channels type. If it is a digital module, the data is stored as a bitmap. For every 8 inputs a byte is needed. For all other modules a data item is a 32-bit signed integer.

bus: ***Select the bus to be used.***

modules: ***Select a module within a bus.***

data: ***Pointer to an array where the data will be stored***

len: ***Size of the array data points at***

return: ***Number of bytes copied into the array data points at***

*INTEGER IOVecOut (UNSIGNED bus, UNSIGNED module, void *data, UNSIGNED len);*

Write all channels of a module with one call. »data« points to an array where the values are taken from. The size of an array item is depending on the channels type. If it is a digital module, the data is stored as a bitmap. For every 8 outputs a byte is needed. For all other modules a data item is a 32-bit signed integer.

bus: ***Select the bus to be used.***

modules: ***Select a module within a bus.***

data: ***Pointer to an array where the data will be stored***

len: ***Size of the array data points at***

return: ***Number of bytes copied into the array data points at***

void IOWD (UNSIGNED bus, UNSIGNED module);

With the TSM system, all outputs are controlled by watchdog timers. It must be guaranteed that all outputs are frequently written. The watchdog period is about 80ms with TSM.

A call to IOWD updates the outputs of a module connected to a specific bus.

6. LINTAB.EXE

LINTAB is a program to calculate linearisation tables for PT100 and Thermocouple sensors. The tables can be used by ExpressIO^{TM} Analog input devices to linearize and to scale analog values to application specific physical units. However, the V1.0 of ExpressIO^{TM} supports PT100 and Thermocouple tables only (The TSM-8AD8<TCK does not need linearisation because it is already linear within the given range and resolution).

There are a lot of parameters - but most are easy to understand.

Steps

Table resolution. The bigger as a table is the more precise are the results. LINTAB will not generate tables with identical table entries, so you can't create uneconomically huge tables! For most applications a step of 5 or 10 works fine. To get optimized tables, steps should be of a size that min. and max. temperature can be divided by steps without remainder.

From

Lower temperature boundary. The possible range is -273...+2000 °C.

To

Upper temperature boundary. The possible range is 1...2274 °C

Zero

DC offset of the sensor in °C. The PT100V4 is a good example for the use of ZERO. Zero is the temperature where the ad-input voltage is 0V! For PT100V4 it is -50 °C. For thermocouples it is usually 0V and zero is not used.

Tname

Name of the LINCTRL structure, the only name exported by the generated file.

Size

The size of the array values. Default (and the only type ExpressIO^{TM} can handle by now) is 16-Bit signed. If size is set to »long« the table consists of 32-Bit signed integers.

Factor

The final scaling factor. It is a signed integer (16-Bit). For example the factor is »10« for all ExpressIO^{TM} temperature inputs (d°C).

Gain

The gain of the sensor conditioner [0.10000...100000]. For PT100 sensors gain is calculated by the formula:

$$\text{gain} = \text{gain}_{amp} * I_{rtd} \text{ [mA]}$$

Range

Voltage range of the raw analog input [0.1...30 Volt]. This is the range from lowest to highest possible voltage. As an example, a +/- 10V input has a range of 20V.

Rawres

Raw value resolution [0.0001...10000]. This value is a pre-calculated value from Res, Gain and Range. Either set »rawres« or the parmeter group [gain,res,range] ... but not both! The only exception is that you are allowed to combine rawres and res for automatic range checking of the ADC - see »res« for more information.

Rawres = (range * 10^6 [µV]) / (2^{res} * gain)

Res

The raw digital resolution in digits (Bits) [4...32] is needed to calculate rawres from gain and range *and* to check if the input value range will not cause an overflow of the ADC.

<type>

For Thermocouples: R, S, B, J, T, E, K

For PT100 (RTD 100Ohm) PT100

Signed

Tells LINTAB that the ADC supplies a signed binary value. This information is used to check for a table overflow.

Examples for using LINTAB:

Example 1:

TSM8AD12 with PT100V4 (-50...205°C) sensor conditioning unit

```
c: > lintab steps=10 from=-50 to=210 gain=51.456 range=5 res=12 zero=-50
        tname=pt100v4 PT100  > myfile.c
        OR
c: > lintab steps=10 from=-50 to=210 rawres=23.723 zero=-50
        tname=pt100v4 PT100  > myfile.c
```

zero=-50:	0V = -50°C
res=12:	12-Bit resolution (8AD12)
range=5:	Voltage range is 0...5V
gain=51.456:	gain: 51.456 = $\text{gain}_{PT100V4}$ * I_{RTD} = 201 * 0.256(mA)
rawres=23.772:	raw resolution = (range * 10^6 [µV]) / (2^{res} * gain) 23.723 = 5 * 10^6 [µV] / (4096 * 51.456)
tname=pt100v4:	name of the LINCTRL structure exported by the generated file
myfile.c:	the c file to be generated. This file can be compiled and linked to the users project.

Example 2: Demonstrate a table overflow!

TSM-8AD12 with external Thermocouple amplifier:

gain_{amp} = 1000

Thermocouple type »K«:

(-200...1300°C, input voltage range with gain = 1000: -5.98..+52.39V)

Voltage input range of ADC = -5...+5V

```
c: > lintab steps=10 from=-200 to=1300 rawres=2.44141
        tname=thermok K  > tck_k.c
c: > lintab steps=10 from=-200 to=1300 gain=1000 range=10 res=12 signed
        tname=thermok K  > tck_k.c
```

res=12:	12-Bit resolution (8AD12)
range=10:	Voltage range is 0...10V (-5...+5V)
signed:	-2048..+2047
gain=1000:	1000
rawres=2.44141:	(range * 10^6 [µV]) / (2^{res} * gain)

$$2.44141 = 10 * 10^{6} [\mu V] / (4096 * 1000)$$

Please note that the table will overflow - only the temperatures between -150..+120 are in range of a 12-Bit AD converter with a ±5V input range! A warning will be generated for every value that is not in range - if res is provided. The following table will fit better:

```
c: > lintab steps=10 from=-150 to=120 gain=1000 range=10 res=12 signed

          tname=thermo_k K  > tck_k.c
```

VI. mCAT Socket Interface

1. Introduction

1.1. Features

The mCAT Socket-Interface is designed to support the *mCAT operating system's* message passing structure. The communication between the protocol layers (TCP,UDP,IP) and an users application task is established, maintained and terminated by exchanging a small set of mCAT-messages, called *Socket Events*. There is also a set of functions to wrap around the naked message passing interface and to hide both complexity and internal structure to the user. First was made to make handling of the socket interface easier to use. Second was made to be able to alter internal structures in future versions of the *mCAT SOCKET LAYER* without changing the user interface and its handling.

Adding socket handling to an application is adding a *socket interface*. Because the socket interface uses generic mCAT message passing, a socket interface can easily be added to existing applications.

Last, not least, the mCAT socket layer is designed to be fast.

1.2. Difference to UNIX Sockets

UNIX (LINUX et.al.) is a classical 1970's operating system. At that time *event driven programming,* as used in modern operating systems like Mac OS, Windows, QNX and – *of course* – mCAT, was not used very frequently. So the basic programming interfaces of UNIX do not support event driven programming style as a natural choice. Even if its possible to do event programming (using *select()* or *poll()*), many socket applications do not use this.

The other concept of UNIX is to presume that all data exchange is done by *streams* of data rather than by well defined blocks (*messages*).

Finally, the UNIX programming interfaces carry a lot of historical burden.

So implementing a UNIX style API on a modern operating systems will need the host operating system to be converted into some type of UNIX. Doing this, it will produce more trouble than benefit.

The *mCAT socket layer* is a modern, easy to use, message passing based socket interface.

2. Socket-Interface

2.1. The Structure of the mCAT TCP/UDP/IP Protocol Stack

The mCAT TCP/UDP/IP protocol stack is implemented in a single system task. The interface to the network device drivers (for example the ETHERNET interface eth0) is included as well as the application interface layer, called *mCAT socket layer (“MSOCKET”)*. The stack supports both TCP and UDP applications.

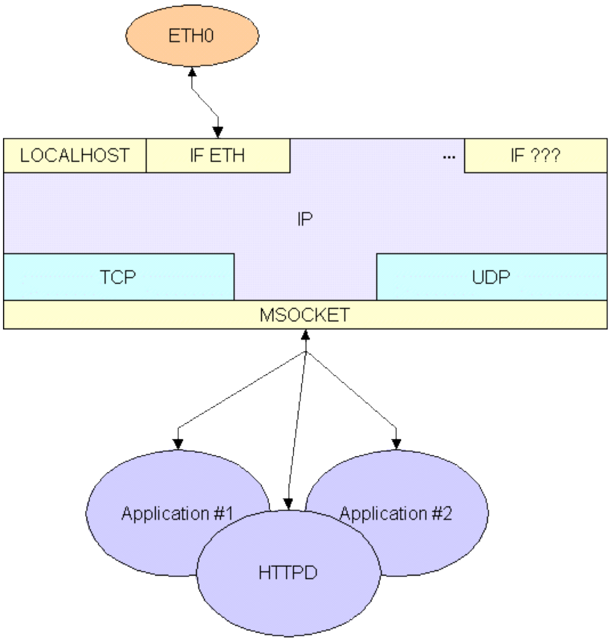


Figure 22: The IP/TCP/UDP protocol Stack

This chapter will give a step-by-step description of the structures and functions needed to add TCP/UDP communication to an application.

2.2. Basic Setup

2.2.1. Before the First Steps

This is not an introduction to mCAT or to the TCP/UDP/IP basics. For mCAT beginners, read the *mCAT Users Manual* and the *mCAT Kernel Reference* first. For TCP/UDP/IP beginners, we recommend *Commer, Douglas E. And Stevens, David L.: Internetworking with TCP/IP, Volume I*.

2.2.2. Ethernet Setup

Usually you will not need to change the setup of the Ethernet interface. However, sometimes it can be helpful to set the interface to fixed mode:

- to increase startup time (auto-negotiation takes some time)
- to limit throughput
- to limit power dissipation (100 mBit consumes much more power than 10mBit)

The EEPROM cell *EE_ETHERNET_MODE* (19) is used to set the ETHERNET interface to a fixed mode. Possible settings are:

AUTONEGOTIATION	1	(or 0xFFFF, default)
10-BASE-T FULL DUPLEX	2	
10-BASE-T HALF DUPLEX	3	
100-BASE-TX FULL DUPLEX	4	
100-BASE-TX HALF DUPLEX	5	
AUTONEGOTIATION10	8	(default on NETA7)
AUTONEGOTIATION100	9	

Starting with R00409 you can use the more convenient SYSMON command *ethmode* to manipulate and display this value.

2.2.3. Create a MSGID

mCAT message communication needs a *message id* (MSGID) structure to route messages from a client to a server. From an mCAT point of view the application is a server to the socket layer – no matter if it acts as a UDP/TCP server or client. Therefore an application needs to define a unique MSGID the socket layer will use to address the application.

A MSGID is created in the *TaskInit()* function of an mCAT application, just after creating the main task. Use the standard mCAT call *MsgIdCreate()* to create the MSGID you need. Choose a unique name for your MSGID.

There are only few rules for MSGID-Names. In general, you should start your mCAT names (both Task-Names and MSGID-Names) with a uniform company or name specific id. Examples:

Company ACME, Programmer Paul More, Application 'TankDetector':

“ACME/PM/TankDetector/SocketInterface”

Call:

```
      socket_interface =
MsgIdCreate(Self,”ACME/PM/TankDetector/SocketInterface”,NULL);
      [note: the length of the id string is not limited with MSGID's]
```

2.2.4. Create a Socket

In the main function of an mCAT application, just before entering the central message loop, a socket must be created. The function *MSCreate(protocol,port,endpoint,listen)* needs four arguments to create a socket:

- a *protocol* identifier. Currently IPT_TCP and IPT_UDP are the only valid values for this argument.
- a *port* number. If K is -1 or 0, an anonymous port number is allocated automatically.
- an *endpoint* for the communication. This is nothing else but the MSGID *socket_interface* we just created.
- for a server socket, *listen* tells the function how many connections (TCP) it is allowed to handle at a time. Please note that this number can not be more than the number of sockets totally available less one.

We can break down the variations of arguments into 4 classes:

- UDP Server: protocol is IPT_UDP, port must be a known number, endpoint is our MSGID and listen is 0.
- UDP Client: protocol is IPT_UDP, port can be -1, endpoint is our MSGID and listen is 0.
- TCP Server: protocol is IPT_TCP, port must be a known number (greater than 0), endpoint is our MSGID and listen must be > 0.
- TCP Client: protocol is IPT_TCP, port can be -1, endpoint is our MSGID and listen is 0.

If the returned socket number (an integer) is less than 0, the function call failed. If the returned value is passed to the function MSGetErrorStr(), a readable error message is returned.

2.2.5. Resolve Domain Names

If the application acts as a client, we need to know the servers *IP-Address* as well as the servers port number to contact it. The port number is usually a known number, maybe even a *well known number*[3]. The IP-Address can be given statically, too. If your server is located in a local area network with a static address mapping, this is a good solution. However, if your server is located on the Internet, it may be difficult to use a static IP-Address. On the Internet, IP-Addresses are changed from time to time without notification. So its better to contact your favourite *domain name server* to resolve the current IP-Address. Therefore the function *MSGetHostByName()* is provided. Use the full domain name as an argument and get the domains IP-Address as a return value.

IMPORTANT: To use this function the IP-Address of the domain name server must be known to the IP-Stack. Refer to section 4.2. Setup IP-Addresses: setip to set IP-Addresses.

2.2.6. The Message Loop

Now we have a socket and an IP-Address. Usually this is all we need to enter the standard mCAT message loop. Socket events are handled in this loop as any other mCAT messages (including BITBUS, TICKER or any other). We know the incoming message is a socket event if the mCAT message header field *type* is equal to the *id* field of the MSGID structure we passed to the *MSCreate()* call. However, there are major differences between an UDP communication and a TCP communication. That is because TCP is a connection orientated protocol, UDP is not. With TCP you have to handle connect and disconnect procedures. With UDP you don't. With TCP you may guard a connection by keep-a-life messages. UDP does not worry.

Lets look a bit more detailed at how UDP and TCP handle their data exchange.

[3] *Well known numbers* are commonly used port numbers. For example, the default well known port number for a *http server* is *80.* A complete list of those well known ports can be found at http://www.iana.org/assignments/port-numbers

2.3. Data Exchange

2.3.1. Connection Less Protocols (UDP)

2.3.1.1. No Connection!

The main feature of *connectionless* protocols is that there is no need to do some initial message exchange *by the protocol* to synchronise a client and a server. The client can send data at any time and frequency. The server can reply at any time and frequency. Nothing is known about the state at the other end of the communication. If you need to know the state at the other end, you have to implement a status exchange under your own control on top of the protocol.

Its very easy to use a connectionless protocol to communicate with more than one server and even between clients. Even the strict terms *client* and *server* can loose their meaning in a connectionless environment. I can send something to Peter, Paul and Mary. They may answer me. They may send me messages that have nothing to do with the messages I sent them.

Connectionless protocols have only a little overhead. The messages sent are transported via the IP/Ethernet without splitting them up into blocks.

However, this class of protocols – and UDP is a very basic implementation of a connectionless protocol – lacks some important features: Flow control, end-to-end data delivery and missing of any state information about the communication counterpart are the most dangerous. Even more confusing: A communication counterpart may not receive the datagrams in the same sequence they have been sent.

2.3.1.2. Be Careful!

- *Do not send too many UDP-messages at a time or too fast after each other. The network may not be able to handle that much traffic!*
- *Do not send bursts of many messages to a single host. It may not be able to handle them!*
- *Do not presume a message you send will be properly received by your counterpart!*
- *Use timeout control at application level to guard your communication!*

- *Add a sequence count to each application message you send (if possible). If an overload situation occurs, any UPD/IP stack will simply drop the extra messages without notification!*

2.3.1.3. Data Exchange

The data exchange is pretty straightforward. *MSWrite()* is used to send data using UDP and the only event that must be handled in the message loop is *MS_EVENT_DATA_AVAILABLE*. With this event, call *MSRead()* to copy the received data into your application's data space. All other socket events must be dropped. Therefore a specific call is supplied called *MSDrop(event)*.

Because UDP datagrams can be sent to any IP-host and the incoming datagrams can be from any other host, *MSRead* and *MSWrite* need to pass address information from / to the socket layer:

```
INTEGER MSWrite(INTEGER socket, void *data, INTEGER dsize, UDPADDR *to, UNSIGNED
urgent);
```

- *socket* refers to the socket we created.
- *data* is a pointer to the data buffer holding the data to be sent.
- *dsize* is the size in bytes of our data buffer.
- *to* is a pointer to a UDPADDR data structure. This structure holds the IP-Address (ipaddr) and the port number (port) the data shall be send to.

```
typedef struct {
    UINT32      ipaddr;             // ip address of src/dest
    UNSIGNED    port;               // port id of src/dest
} UDPADDR;

INTEGER MSRead (MSEVENT *event, void *data, INTEGER dsize, UDPADDR *from);
```

- *event* is a pointer to the socket event received.
- *data* is a pointer to the data buffer that shall hold the received datagram.
- *dsize* is the size in bytes of our data buffer.
- *from* is a pointer to a UDPADDR structure where the address and the port number of the originator of the incoming data will be stored.

2.3.1.4. Socket or Event: A Note on the Referred Objects

As you may have noticed before, MSRead and MSWrite refer to two totally different datatypes: MSWrite to the *socket* and MSRead to an *event*. That has to be explained.

Functions that accept an *MSEVENT* as a first argument are designed to handle events sent by the mCAT *socket interface layer*. All information needed to handle such events are included in the events structure.

Functions that accept a *SOCKET IDENTIFIER* (an integer > 0) as a first argument can be called at any time. They usually do not depend on the information carried by events. The *socket identifier* is a supplied by the *MSCreate()* call or by a *MS_EVENT_CONNECT* event.

2.3.2. Connection Orientated Protocols (TCP)

2.3.2.1. Establish a Connection

Using connection orientated protocols (TCP in our case) has several advantages. Before the data exchange can be issued, a connection must be established. TCP uses a sequence of system messages to contact the communication counterpart and to negotiate the communication parameter. So if a connection is established once, we can be sure our communication counterpart is reachable and available.

When data is exchanged, both communication partners will trace the number of data bytes exchanged, will take care that data is delivered in the correct sequence and that no data is delivered twice or lost.

It's a very good idea to compare a connection orientated protocol to a telephone call. First you have to dial a number. When your counterpart picks up his phone the connection is established. Once established, you can use the connection to talk until everything is said and you or your counterpart decides to hang up. This is the disconnect.

The disadvantage of a connection orientated protocol is the need to keep track of all open connections. That is in fact important for a TCP-Server, because the server should be able to serve more than one connection at a time.

2.3.2.1.1. mCAT Application as a TCP Server

To simplify the handling of connections, the mCAT-API offers servers a feature called *application context identifier (acid).* This is an user defined integer that is passed to the TCP protocol stack when a connection is accepted. Whenever data exchange or disconnect events arrive at the server application, the protocol stack passes the *acid* along with the event. So the application can identify to which open connection the current event belongs to. See 4.3. A simple mCAT-TCP-Server for an example.

The server side of the TCP-Protocol and the socket interface is the most complex part of the entire TCP/IP package. The server application creates one socket that is bound to a port number known by the clients or bound to a *well known port number*. This socket will never be used for data exchange. It is used for connection handling ONLY! When a client connects to a server and the connection is accepted, a new, hidden and anonymous socket is created by the *socket layer interface* to handle this new connection (one socket can handle one connection only). The *socket identifier* of this new socket is passed along with the *MS_EVENT_CONNECT* event and can be taken from the events body before the connection request is accepted.

It is important to note that this is not a feature specific to the mCAT-Socket-API. The mechanism is the same as with UNIX or WINDOWS sockets.

2.3.2.1.2. mCAT Application as a TCP Client

The client side is straight forward. You need to create one socket for each connection. The port number of the client is of no interest, it can be an automatically allocated anonymous port number. If more than one connection must be served, using different MSGID's make it easy to relate an event to its socket.

2.3.2.2. Data Exchange

Once a connection is established, the data exchange is pretty close to the connectionless communication – except that now a powerful protocol takes responsibility for the secure data transport. In contrast to the connectionless data communication the *MSRead* and *MSWrite* do not need references to address information (UDPADDR *) - passing a NULL pointer instead of pointers to address structures is a good practice.

2.3.2.3. Terminating a Connection

Only a client can open a connection to a server – a server can not open a connection. But both ends can disconnect a connection at any time and at any state of communication. Again, the analogy to a telephone call is evident. Both partners can hang up at any time.

One option to disconnect is to close the socket. This is a rude way to give up a connection. So the proper option is to use the *MSDisconnect()* call.

Please note that sockets that have been created by the socket layer to serve a connection (server) are also closed automatically when the related connection is disconnected.

2.3.2.4. Keep-Alive

One disadvantage of the original TCP-protocol is that if a connection is established once, it may stay connected forever without notice. That is because if there is no data exchange, there are no control messages to make sure the connection is still open/alive and the counterpart is reachable.

To fix that, a server may be configured to send simple protocol messages on idle. Those messages are called “*keep-alive*” messages. The mCAT-Socket-API supports *keep-alive* by means of the API function called *MSKeepAlive()*. It is highly recommended to keep connections not open for a longer time. But if it is necessary to keep a connection open, use the *keep-alive* feature to guard your connection.

2.4. Utility Functions

The mCAT-Socket-API supports application design by a set of small helper routines, like conversion routines for IP addresses. Please refer to 3.4. Miscellaneous for details.

3. The Function Reference

3.1. Create and Close Sockets

```
INTEGER MSCreate(INTEGER protocol,INTEGER port,MSGID *endpoint,INTEGER listen);
```

Function: Create a socket.

Arguments:

protocol IPT_TCP or IPT_UDP

port The port number to be associated with this socket. 0 or -1 will allocate an anonymous port number (suitable for clients).

endpoint A pointer to a MSGID structure. The socket interface layer will send all socket events related to this socket to the msgid *endpoint*.

listen 0 for a TCP client and for UDP applications. If listen is > 0, its the number of connections a TCP server may handle at a time.

Returns:

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

```
void MSClose(INTEGER socket);
```

Function: Close a socket.

Arguments:

socket The socket identifier of the socket to be closed. Open connections are closed automatically.

Returns: -/-

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

3.2. Handling Connections

```
INTEGER MSConnect(INTEGER socket,UDPADDR *dest);
```

Function: Used by a client to establish a connection to a server. If the server is reachable and if it accepts the connection, MS_ERROR_OK is returned.

Arguments:

socket	Refers to the client socket created by an MSCreate() call.
dest	Pointer to a UDPADDR structure. This structure is used to pass the internet address (member *ipaddr*) and the port number (member *port*) of the server to be contacted.

Returns: MS_ERROR_OK if connection is established.

Any other value signals an error. See 3.6. Socket error codes for more information on error codes.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments: This call can be used by a client only.

```
INTEGER MSAccept(MSEVENT *event,MSGID *endpoint,UNSIGNED acid);
```

Function: Used by a server to accept a request by the client to establish a connection. The call provide some management data to the TCP-protocol stack that is needed to handle the Data exchange.

Arguments:

event Pointer to the socket API event *MS_EVENT_CONNECT*

Note: The socket identifier of the new anonymous socket created to serve this connection is passed in the event structures member *socket.*

endpoint An mCAT Message ID to used with the new connection. Please note that a new, anonymous socket is created when a connection is established. The new socket needs a MSGID for the same purpose as a socket created by a *MSCreate()* call.

You may use the same MSGID as you used in the MSCreate() call but you may also use a different one.

acid The *application context identifier.* The value passed via this argument will be passed by the socket interface to with every future socket event that belongs to this connection. For example, this may be an *array index* into your connection status table.

Acid is passed in the events structures member of the same name.

Returns: MS_ERROR_OK if no error occured.

Any other value signals an error. See 3.6. Socket error codes for more information on error codes.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

```
INTEGER MSRefuse(MSEVENT *event);
```

Function: If a connection request should not be accepted by a server (for what reasons ever) it can call MSRefuse() to refuse the request.

Arguments: event Pointer to the socket API event *MS_EVENT_CONNECT*

Returns: MS_ERROR_OK if connection is established.

Any other value signals an error. See 3.6. Socket error codes for more information on error codes.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

```
INTEGER MSDisconnect(INTEGER socket);
```

Function: A client can call *MSDisconnect()* at any time or state of communication to terminate a connection. The server will be informed about the disconnection by mean of a *MS_EVENT_CONNECT* event.

Arguments: socket Refers to the client socket created by a MSCreate() call.

Returns: MS_ERROR_OK if connection is established.

Any other value signals an error. See 3.6. Socket error codes for more information on error codes.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments: This call can be used by both client and server.

```
INTEGER MSAckDisconnect(MSEVENT *event);
```

Function: When a server receives a *MS_EVENT_DISCONNECT* event, it can release all status information associated with the referred connection. However, it MUST call *MSAckDisconnect()* to acknowledge the disconnect.

Arguments: event Pointer to the socket API event *MS_EVENT_DISCONNECT*

Returns: MS_ERROR_OK if connection is established.

Any other value signals an error. See 3.6. Socket error codes for more information on error codes.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

```
void MSSetKeepAlive(INTEGER socket,INT32 timeout);
```

Function: This call is used on the server side only. It configures the TCP protocol stack to send all *timeout* milliseconds a keep-alive message. A suitable value for timeout is 60000 (1 minute) in most applications.

Arguments:

socket Refers to the client socket created by a MSCreate() call.

timeout A timeout value in milliseconds. 0 disables sending keep-alives.

Returns: -/-

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments: Please note that “traditional” TCP/IP books say that the keep-alive timeout should be not less than 2 hours to prevent senseless traffic. That may be suitable for UNIX servers, where hundreds of sockets and connections can be used. But this is fatal with small embedded controllers where you can have maybe 16 connections at all. If one or more connections are blocked because the system does not notice that the communication counterpart was removed from the network, the function and performance of the node may not satisfy customers needs.

3.3. Data Exchange

```
INTEGER MSRead(MSEVENT *event,void *data,INTEGER dsize,UDPADDR *from);
```

Function: This function is used to copy the received TCP or UDP data into the receivers data space. The data are passed to an application by use of an *MS_EVENT_DATA_AVAILABLE* event.

Please note that this call also frees the event, you can't read data from one event more than once.

Arguments:

event Pointer to the socket API event *MS_EVENT_DATA_AVAILABLE*

data Pointer to a buffer to store

dsize Size of the user buffer. Should be not shorter than the networks MTU. 1500 is a good choice for the buffer size.

from Used in a UDP application to receive the counterparts IP-address and port number. Can be NULL if not needed.

Returns: If not negative, the number of bytes transferred to the buffer.

Any other value signals an error. See 3.6. Socket error codes for more information on error codes.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

```
INTEGER MSWrite(INTEGER socket,void *data,INTEGER dsize,UDPADDR *to,UNSIGNED urgent);
```

Function: Used to send data. With TCP, the data is send via an already established connection. With UDP, the data is send and a single datagram to the address specified by argument *to*.

Arguments:

socket Refers to the socket -

- created by an MSCreate() call if its a client
- created by an MSCreate() call if its a UDP server
- provided along with the *MS_EVENT_CONNECT* if it is a TCP server. See also *MSAccept()*.

data Pointer to the data to be send

dsize Size data to be send in bytes. Should be not longer than the networks MTU. However, the call returns the number of bytes sent, so a further call can be issued to send residual data.

to In an UDP application use *to* to pass the counterparts IP-address and port number to the UDP protocol stack.

Can be NULL in TCP applications.

urgent With TCP, pass a non negative value if you want to send an *urgent value*. Usually, TCP designers try to prevent using of the urgent feature, because it is not well defined.

No meaning in UDP.

Returns: If not negative, the number of bytes sent.

Any other value signals an error. See 3.6. Socket error codes for more information on error codes.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

3.4. Miscellaneous

```
INTEGER MSDrop(MSEVENT *event);
```

Function: Every socket event that is NOT used shall be dropped by use of *MSDrop().* MSDrop takes care of events and maybe buffers being associated with the current event are freed properly.

Arguments: event Pointer to the socket API event
MS_EVENT_DATA_AVAILABLE

Returns: MS_ERROR_OK if connection is established.

Any other value signals an error. See 3.6. Socket error codes for more information on error codes.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

```
IPADDR MSGetHostByName(char *name);
```

Function: This function is used to resolve a host name (www.mocom-software.de). If a *domain name server* address is configured, it is reachable and the host name is known, the call will return the a proper IP address.

This function can also be used to convert a visible string holding an IP address in the format “xxx.xxx.xxx.xxx” into a binary IP address with the correct endian.

Arguments: name A string containing the host name or a “xxx.xxx.xxx.xxx” IP address.

Returns: A valid IP address or 0 if failed.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments: ipaddr = MSGetHostByName(“www.mocom-software.de”);

or

ipaddr = MSGetHostByName(“172.31.31.54”);

```
INTEGER MSIP2String(IPADDR ip,char *buffer,UNSIGNED len);
```

Function: Converts a given IP address into a visible string representation.

Arguments:

ip	The IP address to convert
buffer	A buffer to store the visible string
len	Length of the buffer. Minimum length is 16 bytes.

Returns: FALSE if len was too small.

If the IP address 0x351F1FAC is passed to this function, buffer will hold the string “172.31.31.53” after the call.

Note that IP addresses are store in the big endian format only.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

```
char * MSGetErrorStr(UNSIGNED error);
```

Function: Returns a visible representation of an error code

Arguments:

error	A MS_ERROR_???? error code

Returns: A string representing the code.

If MS_ERROR_REFUSE is passed to *IPGetErrorStr()*, the string “CONNECTION REFUSED” will be returned.

Please note that if error code is greater than 0, "OK" will be returned.

Supported:

mCAT	2.20 (ARM) and higher
Hardware	All with ETHERNET interface

Comments:

3.5. The Socket Events

The structure of a socket event is pretty complex. But there are only a few members that are of interest for an application developer.

```
typedef struct {
    MSG         hdr;                // std hdr
    INTEGER     cmd;                // command code (MS_EVENT_...)
    INTEGER     error;              // error code (currently not used)
    INTEGER     socket;             // socket id
    UNSIGNED    acid;               // application context id
    ...
} MSEVENT;
```

All other members should only be handled indirectly using API calls!

3.5.1. MS_EVENT_DATA_AVAILABLE

Informs an application that data is available. Use the *MSRead()* call to transfer data into the applications data space and to free buffers associated with this event.

3.5.2. MS_EVENT_CONNECT

A TCP server receives this event if a client tries to connect. Use *MSAccept()* to accept or *MSRefuse()* to refuse the request.

3.5.3. MS_EVENT_DISCONNECT

Use to inform an application that a TCP connection is to be disconnected. Must be acknowledged using *MSAckDisconnect()*.

3.6. Socket Error Codes

```
    MS_ERROR_OK=0                       OK, no error
    MS_ERROR_SYS=-1                     Unspecified system error
    MS_ERROR_REFUSE=-2                  Connection request was refused by server
    MS_ERROR_MEMORY_OVERFLOW=-3         Memory overflow
    MS_ERROR_CONNECTION_BROKEN=-4       Connection broken
    MS_ERROR_UNKNOWN_PROTO=-5           Unknown protocol request
    MS_ERROR_PORT_INUSE=-6               Port is occupied
    MS_ERROR_NO_SOCKET=-7               Invalid socket id passed
    MS_ERROR_CONNECTIONLESS=-8          Tried to "connect" a non tcp port
    MS_ERROR_USER_BUFFER=-9             User buffer too small!
    MS_ERROR_NOT_OPEN=-10               Connection no opened yet
    MS_ERROR_LISTENING=-11              MSConnect can only be called from a client
    MS_ERROR_DESTINATION_NOT_REACHABLE=-12
```

4. Examples

4.1. GetTime using UDP

```

INTEGER TaskInit (IMD *imd)
{
    INTEGER error;
    Protect();
    Self = TaskStartup(imd,FromTop,&error,0);
    sockid = MsgIdCreate(Self,"MY/UDP/Socket",NULL);
    UnProtect();
    return Self;
}

INTEGER TaskMain (void)
{
    MSGID       *id_ticker;
    MSEVENT     *evt;
    INTEGER     socket,
                len;
    UDPADDR     daytime;
    UINT64      start,end;

    kprintf("*** udp daytime test ***\n");

    daytime.ipaddr = MSGetHostByName("ptbtime1.ptb.de");
    if (daytime.ipaddr == 0) {
        kprintf("DNS fail\n");
        daytime.ipaddr = MSGetHostByName("172.31.31.254");
    } /* endif */

    // ip addresses are ALLWAYS in big endian format
    strcpy(buffer,"---.---.---.---");
    MSIP2String(daytime.ipaddr,buffer,39);
    kprintf("SERVER=%s\n",buffer);

    // setup destination port
    daytime.port = 13;

    // Create Socket
    socket = MSCreate(IPT_UDP,0,sockid,0);
    if (socket <= 0) {
        kprintf("SOCKET ERROR=%s\n",MSGetErrorStr(socket));
        TaskDelete(Self);
    } /* endif */

    id_ticker = TALL(&tick,1000l,128,Self);
    kprintf("SELF=%d ticker=%8.8lx\n",Self,id_ticker->id);

    // send UDP Query
    MSWrite(socket,"what time is it?\n",sizeof("what time is it?\n"),&daytime,0);
    // handle
    loop {
        evt = MsgWait(0,SYS_WAIT_INFINITE);
        if (evt) {
            if (evt->hdr.type == sockid->id) {
                switch (evt->cmd) {
                  case MS_EVENT_DATA_AVAILABLE:
                    len = MSRead(evt,buffer,512,NULL);
                    end = 0;
                    Time64SinceStart(&end);
                    if (len <= 0) {
                        kprintf("Error reading data %s\n",MSGetErrorStr(len));
                    } else {
                        buffer[len] = 0;
                        kprintf("DAYTIME[%d]=%s [%Ld]\n",len,buffer,end-start);
                    } /* endif */
                    break;

                  default:
                    MSDrop(evt);
                } /* endswitch */
            } else if (evt->hdr.type == id_ticker->id) {
                MsgSendReply(evt,Self,ACK);
                start = 0;
                Time64SinceStart(&start);
                MSWrite(socket,"what time is it?\n",sizeof("what time is
it?\n"),&daytime,0);
            } else {
                kprintf("UNKNOWN\n");
            } /* endif */
        } else {
            // timeout
            kprintf("TOUT\n");
            MSWrite(socket,"what time is it?\n",sizeof("what time is
it?\n"),&daytime,0);
        } /* endif */
    } /* endloop */
}
```

4.2. GetTime using TCP

```
INTEGER TaskMain (void)
{
    MSGID       *id_ticker;
    MSEVENT     *evt;
    INTEGER     socket,
                len;
    UDPADDR     daytime;
    int         error;

    kprintf("*** tcp daytime test ***\n");

    daytime.ipaddr = MSGetHostByName("ptbtime1.ptb.de");
    if (daytime.ipaddr == 0) {
        kprintf("DNS fail\n");
        daytime.ipaddr = MSGetHostByName("172.31.31.254");
    } /* endif */

    // ip addresses are ALLWAYS in big endian format
    strcpy(buffer,"---.---.---.---");
    MSIP2String(daytime.ipaddr,buffer,39);
    kprintf("SERVER=%s\n",buffer);

    // setup destination port
    daytime.port = 13;

    // Create TCP-Client Socket
    socket = MSCreate(IPT_TCP,0,sockid,0);
    if (socket <= 0) {
        kprintf("SOCKET CREATION ERROR=%s\n",MSGetErrorStr(socket));
        TaskDelete(Self);
    } /* endif */

    error = MSConnect(socket,&daytime);
    if (error != MS_ERROR_OK) {
        PError(__LINE__,error);
        TaskDelete(Self);
    } /* endif */

    id_ticker = TALL(&tick,10000l,128,Self);
    kprintf("SELF=%d ticker=%8.8lx\n",Self,id_ticker->id);

    MSWrite(socket,"what time is it?\n",sizeof("what time is it?\n"),NULL,-1);
    // handle
    loop {
        evt = MsgWait(0,SYS_WAIT_INFINITE);
        if (evt && evt->hdr.type == sockid->id) {
            switch (evt->cmd) {
              case MS_EVENT_DATA_AVAILABLE:
                len = MSRead(evt,&buffer,sizeof(buffer),NULL);
                if (len < 0) {
                    PError(__LINE__,len);
                } else {
                    buffer[len-2] = 0;
                    kprintf("TIME='%s'\n",buffer);
                }
                break;

              case MS_EVENT_DISCONNECT:
                MSAckDisconnect(evt);
                break;

              default:
                MSDrop(evt);
            } /* endswitch */
        } else if (evt->hdr.type == id_ticker->id) {
            MsgSendReply(evt,Self,ACK);
            //kprintf("SEND QUERY ..");
            error = MSConnect(socket,&daytime);
            if (error != MS_ERROR_OK) {
                PError(__LINE__,error);
                TaskDelete(Self);
            } /* endif */
            MSWrite(socket,"what time is it?\n",sizeof("what time is
it?\n"),NULL,-1);
        } else {
            kprintf("UNKNOWN\n");
        } /* endif */
    } /* endloop */
}
```

4.3. A Simple mCAT-TCP-Server

```
/*
 *    mcattcpif
 *
 *    (c) 2004 mocom software GmbH & Co KG
 *
 *    File: mcattcpif.c
 *
 *    Created: 21.01.2004  17:29  VG
 *    ----------------------------------------------------------------------
 *
 *    History:
 *
 *     date     time       author       comment
 *    ----------------------------------------------------------------------
 *     21.01.2004  17:29      VG        mCAT TCP INTERFACE
 *    ----------------------------------------------------------------------
 */
#include <mcat.h>
#include <socket.h>
#include <nvmem.h>
#include <nameserv.h>
#include <memmgr.h>
#include <simpleio.h>
#include <eeprom.h>

#define MAX_CONNECTIONS 4

PUBLIC  INTEGER Self;
PRIVATE MSGID   *session;
PRIVATE INTEGER session_alloc[MAX_CONNECTIONS];
PRIVATE char    buffer[1500];

INTEGER TaskInit (IMD *imd)
{
      INT16 error;

      Protect();
      // std. mCAT Task startup
      Self = TaskStartup(imd,FromTop,&error,0);
      // create message ID
      session = MsgIdCreate(Self,"mCAT/TCPIF",NULL);
      UnProtect();

    return Self;
}

void PError(INTEGER line,INTEGER error)
{
    if (error <= MS_ERROR_OK)
        kprintf("SOCKET ERROR [%d]='%s' @%d\n",error,MSGetErrorStr(error),line);
}

INTEGER TaskMain (void)
{
    MSEVENT     *evt;
    INTEGER     socket;
    INTEGER     i,
                error,
                len;

    kprintf("*** mCATTCPIF ***\n");

    // Create Socket, up to MAX_CONNECTIONS connections allowed
    // port number is 9000
    socket = MSCreate(IPT_TCP,9000,session,MAX_CONNECTIONS);
    if (socket <= 0) {
        kprintf("SOCKET ERROR=%s\n",MSGetErrorStr(socket));
        TaskDelete(Self);
    } /* endif */

    // enable TCP keepalive messages, timeout is 60sec
    MSSetKeepAlive(socket,60000);

    // handle
    loop {
       evt = MsgWait(0,SYS_WAIT_INFINITE);
        if (evt && evt->hdr.type == session->id) {
            switch (evt->cmd) {
              // a client wants to open a connection
              case MS_EVENT_CONNECT:
                    // allocate a connection context - a structure or variable
                    // to store information related to this specific connection.
                    // In this simple example, we store just the socket identifi-
er.
                    // Please note that we use a array to store the context and
that
                    // we pass the array index as an ACID back to the socket. Ev-
ery
                    // future event generated by this socket will carry the acid
back
                    // and makes it so very easy to locate the related connection
context!
             for (i=0;i<MAX_CONNECTIONS;i++) {
                 if (session_alloc[i] == 0) {
                     // good, accept this
                     session_alloc[i] = evt->socket;
                     error = MSAccept(evt,session,i);
                     kprintf("CONNECT %d\n",i);
                     break;
                 } /* endif */
             } /* endfor */
                 if (i > MAX_CONNECTIONS-1) {
                     // refuse if too many connections
                        kprintf("APP: REFUSE\n");
                        MSRefuse(evt);
                    } /* endif */
             break;

              // data from the client is available
              case MS_EVENT_DATA_AVAILABLE:
                // data receive
                len = MSRead(evt,buffer,sizeof(buffer),NULL);
                if (len > 0) {
                    // send echo. Very simple demo
                      // But you see the advantage of using the acid as
                      // a table entry!
                    MSWrite(session_alloc[evt->acid],buffer,len,NULL,-1);
                } /* endif */
                break;

              // client wants to disconnect
              case MS_EVENT_DISCONNECT:
                if (session_alloc[evt->acid]) {
                    session_alloc[evt->acid] = 0;
                    kprintf("REMOTE DISCONNECT %d\n",evt->acid);
                    MSAckDisconnect(evt);
                } /* endif */
                break;

              // everything other MSEVENT can and must be ignored by MSDrop()
              default:
                MSDrop(evt);
            } /* endswitch */
        } /* endif */
    } /* endloop */
}
```

5. Limitations

- ICMP supports the PING command only.
- No support for RAW-IP.
- No UDP-Broadcasts supported.
- No DHCP support.

- No SSL support.
- UDP-Message length is limited to the MTU length.

VII. mCAT HTTPD Server

1. Introduction

The mCAT-HTTPD-Server is a powerful embedded HTTPD server with a small memory footprint. In contrast to other embedded servers, the mCAT server can handle multiple connections and it can maintain up to 3 independed servers at a time, each serving an individual TCP port. The included MSP (mCAT Server Page) language is a smart macro language to help users to design easy to maintain pages with a high degree of direct access to system functions and to user supplied C code functions.

All files available through the Server (HTML,GIF's, ...) are stored in a read-only file system in the FLASH memory of the embedded node. A easy to use tool (***HTFSBUILD***) is provided to generate a downloadable image of the file system using a normal Windows® sub-directory tree as a template.

1.1. CGI-Handling

1.1.1. What is CGI-Handling?

CGI is the synonym for “COMMON GATEWAY INTERFACE”. It is a specification on how *arguments* can be passed with a resource request using the HTTP GET or POST methodes. Usually, CGI is used to pass arguments from an HTML-Form to an executable file on the server to process the inputs made to this form.

1.1.2. Traditional CGI-Handling

Traditionally, when passing the results from a form to the server, a so called **CGI-SCRIPT** is executed on the server to process the data and to send an answer to the client. CGI-Scripts can be written in almost any language including PERL, phyton, VB or even C. Usually, the output to the client is hard coded HTML-Code send to the client using a sequence of “printf” like calls. On most systems, CGI-Scripts are located in the *cgi-bin* sub-directory.

There are two disadvantages with this approach:

1. Usually CGI-Scripts mix the data processing and the data representation. That makes it hard to change the “look and feel” without changing vital processing code. If you have to maintain a CGI-based application for different customers using different look and feels, it may be a hard job!

2. In an embedded system there is usually no choice to use memory-wasting and performance-eating script languages. We have no 2GHz CPU. We have no swap-device. So the only alternative is to use C for CGI-Scripts and that can be a piece of hard work if the project is complex.

1.1.3. HTML-Template processing using MSP

To separate data processing and data representation as well as to keep the price in memory consumption and performance low, we introduce a small but powerful template processor. It is a bit like a C-Preprocessor but with more flexibility and more power.

First, be aware that the CGI-Conventions are more general. You can pass CGI-Arguments to *any* file and it is *not a must* to have those files in *cgi-bin*!

It is even possible to hand-code the argument list in an HTML HREF tag:

<A HREF="www.myserver.com/thisfile.htm?test=1&cgi=0">TEST</A>

In this example, the arguments with the names *test* and *cgi* are passed along with their *values* to *www.myserver.com/thisfile.htm.*

So if we have a template processor, that can read, insert, process and/or modify these CGI-arguments and call user defined (or system intrinsic) c-functions to do so, we should be able to handle *many* (if not all) embedded CGI-Applications easily.

Looking from outside, a "normal" HTML page and a CGI-Script differ in nothing but the presence of an argument list. MSP is the technology to handle the arguments in HTML files.

It is a tool for creating embedded dynamic HTML pages.

2. The mCAT-HTTPD-Server setup

2.1. TCP/IP configuration

First you have to make sure that the TCP/IP configuration works fine. Please refer to 1.5. SYSMON-Support in the Socket documentation for details.

2.2. The HT-FileSystem (HTFS)

The HTFS file system is an easy to use read-only file system. The tool *htfsbuild* creates an image file from a normal windows directory tree. This image can be relocated to any address in the target system. The recommended address for HTFS images is 0xA00000-0xAFFFFF (1MByte).

There is one thing to take care of: With HTFS, all file and directory names are changed to lower case. Please be aware of this and use only lower case references in your HTML code!

Using a Windows directory tree as template is pretty comfortable.

- create a directory and create (using one of the examples as a template) a config.db file.
- create a root sub-directory and insert the name of this root into config.db
- copy everything you need (gif, jpg, html, txt) into this directory
- copy a makefile from one of the examples to the base directory and edit the makefile as needed (name of the image tree)
- call vmake to build the image.
- download the generated SHX-File

Please study the example *<mcatpath/cc/httdp/1-simple*. Here you can see the basic operation at a glance!

2.3. Basic Authentication

Starting with Release 402 and htfsbuild 2.12 the mCAT HTTPD-Server supports Basic Authentication. This is not a very hard security mechanism, but it is good enough to protect websites from unauthorized changes by chance - An open door may tempt a saint.

To protect a website, two steps are required:

1. Add user-names and passwords to config.db. See 2.4.1.8. User-Names and Passwords for details.
2. Add a ..protect file to each sub-directory. The ..protect file is a plain text file holding the list of users allowed to read files from this directory. A single '*' char allows all users to access this directory. Please note that if user-names are included into config.db each accessible directory MUST have a ..protect file. If not, its unaccessible via HTTP.

2.4. The 'config.db' file

The config.db is designed to configure your mCAT-HTTPD-Server. It is a plain text file and easy to maintain. Most of the text are MIME file assignments that must be included. HTFS-BUILD will tag all files with the suitable MIME-ID when building the HTFS image. That makes MIME handling in the embedded system very easy. It is recommended to use a config.db from one of the examples as a template to include all MIME definitions.

The basic information you have to edit is the server definition:

```
SERVER=MyWebSite
MyWebSite.port=80
MyWebSite.connections=4
MyWebSite.root=/root
MyWebSite.users=*
MyWebSite.keepalive=60
MyWebSite.txbuffersize=32
MyWebSite.rxbuffersize=32
MyWebSite.mempool=32
```

This definition creates a Webserver named MyWebSite listening to port 80. We allow to maintain up to 4 connections at a time. The HTML-Root directory within the image is “/root”. We agree to allow a TCP keep-a-live time of 60 seconds. The receive and transmit buffers as well as the memory pool for MSP execution are being set up, too.

Note: By now, users is reserved.

The following configuration is a bit more advanced. It setup two servers: “Service” and “View”.

```
SERVER=VIEW,SERVICE

SERVICE.port=8000
SERVICE.connections=1
SERVICE.root=/service
SERVICE.users=*
SERVICE.keepalive=20

VIEW.port=80
VIEW.connections=4
VIEW.root=/service/view
VIEW.users=*
VIEW.keepalive=60
```

2.4.1. Config Parameters

2.4.1.1. Port

You can have up to three instances of the HTTPD server running at a time. Every server needs an unique port number. The parameter ***port*** is used to set the desired port number. Please note that the default port for HTTP is 80. A server listening at a non-standard port can be accessed by adding the port number to the *URL*. For example, a server listening at 8000 can be accessed using a web browser by typing

```
        http://172.31.31.51:8000/index.htm
```

2.4.1.2. Connections

On an embedded system with limited resources it makes no sense to allow an unlimited number of TCP connections to be opened at a time. In some cases it even makes sense to limit the number of connections down to 1. Using the parameter ***connections*** you can configure the mCAT HTTPD as needed.

2.4.1.3. Root

The parameter ***root*** sets the root directory for the server inside the HTFS file system. Root can not be omitted.

There are several ways to use root if more than one HTTPD server shall be used:

1. Give all servers the same root. This will give all servers the access to the same files and the same directory structure.
2. Give them different roots on the same directory level. This will make it impossible for the servers to access the files of another server.
3. Give them different roots within the common directory hierarchy. That will allow them to share resources but will also hide private properties. Please study example “*5-msp*” from the mCAT installation. It shows two websites using this approach.

2.4.1.4. Keepalive

An idle TCP connection does not exchange any messages. If the connection is broken (because someone pulls a plug or a system is crashed), the server will not be notified and the connection will stay open until the system is restarted. To prevent this, TCP defined a keep-a-live mechanism. When no message is received within a given timeout the server will send a *TCP ACK* frame for a packet it already received. The counterpart of the connection will answer this *ACK* by another *ACK.* In case the client responds correctly, the connection is treated as intact and it is kept open. If not, another ACK is send and if this is still not answered, the connection will be closed.

Setting ***keepalive*** to a reasonable value will keep the server accessible. The value is the timeout in seconds. Zero (0) disables the keep-a-live mechanism.

2.4.1.5. Txbuffersize

The transmit buffer is used as a buffer to store data to be send. In case of MSP pages, the entire page must fit into the transmit buffer before it can be send. The default is 8kByte and this is usually enough (this is not true for other files, like graphic image files). But sometimes its not and you can configure your server to use a bigger transmit buffer.

2.4.1.6. Rxbuffersize

The receive buffer is also 8kBytes by default. Usually there is no need to enlarge it. It may be necessary to enlarge the receive buffer if large HTML-Forms have to be processed.

2.4.1.7. Mempool

While an MSP-Page is processed, dynamic memory is needed to store temporary results and data. This is done in a fixed size memory pool. After executing an MSP page, the memory pool is cleared and it is empty for the next MSP-Page processing. Usually, 8kByte are enough for a reasonable complex MSP-Page. However, to be flexible the size of this memory pool can be changed by use of parameter ***mempool***.

2.4.1.8. User-Names and Passwords

To make users and their passwords known to the system, they are included into config.db. User-names and passwords can contain almost all ASCII characters including space (' '). Not allowed are ':', witch is used to separate user names and passwords, ',' witch is used to separate user-name:password groups and the newline character ('\n').

```
USERS=Aladdin:open sesame,willi:sowieso,paul:genau
```

A maximum of 16 users can be assigned. The line in config.db may not exceed 1024 Characters.

3. The mCAT Server Page (MSP) Language

3.1. CGI Argument passing

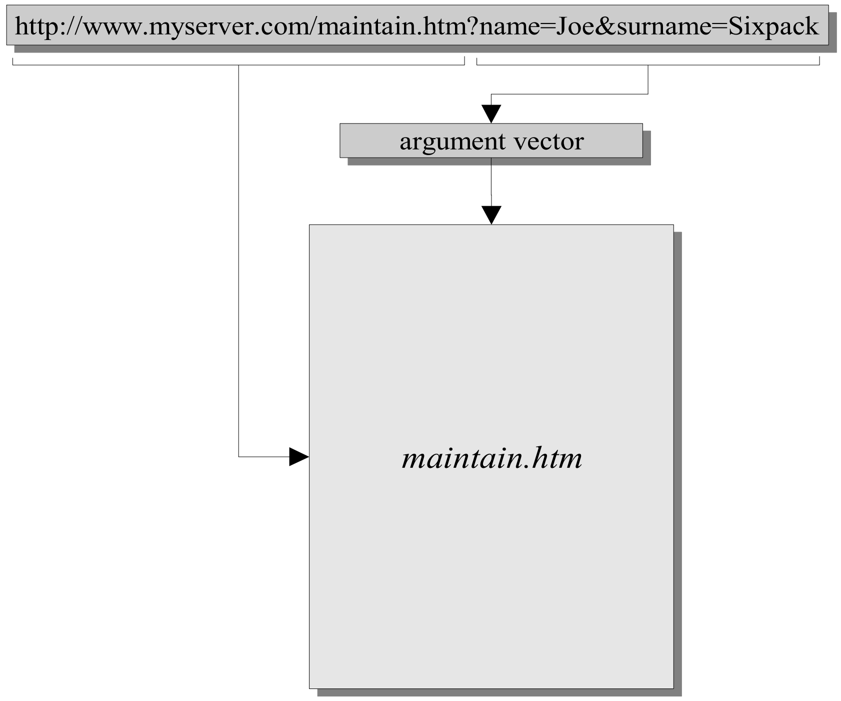

http://www.myserver.com/maintain.htm?name=Joe&surname=Sixpack
argument vector
maintain.htm

Figure 23: URL and argument passing

The CGI-Arguments passed with the URL are parsed by MSP and stored in a standard argument list data structure (see 3.2.6. Argument Lists and 3.5.1. Internal Representation of Argument Lists). There are only a few rules to follow to form a valid CGI-Argument list:

- A question mark ('?') is used to separate the documents URL and the argument list
- The argument list consist of one or more pairs of an argument *name* and an argument *value.*
- If there are more than on *name/value* pair, the pairs are separated by an ampersand ('&').
- If an ampersand, a question mark or any other special character not allowed in an URL (like *space* ' ') occurs in a value, it must be replaced by its hexadecimal representation. The hexadecimal representation is formed by a leading percent sign ('%') and two hex-

adecimal digits. The MSP-Parser replaces hexadecimal values with their original character value. If the percent sign ('%') must be included in a value, it must be replaced by its hexadecimal representation.

An MSP-User has not to deal with all those rules as long as he does not need to form a CGI-Argument list by hand. However, for testing a page it can be helpful to add arguments manually.

Note: The CGI-Specification does not require that arguments passed as name/value pairs. But MSP does! However, the most common use of CGI-Arguments is to pass HTML-Form data and there you usually use name/value pairs to have an unique way to identify data.

3.2. An MSP Statement

An MSP-Statement starts with the MSP-ESCAPE character ('@'). This is immediately followed by another ESCAPE character ('@' or '%'). If not, it is assumed that this is not an MSP statement and the text is send without modification. There are three possible forms of an MSP statement:

1. KEYWORD Statement

   @@keyword{(<argument list>)}

2. Function or Filter call Statement

   @@namespace.key(<argument list>)

3. Constant Reference

   @%constant_name

3.2.1. Charsets

HTML files containing MSP statements must be of ASCII or UTF8 format. Other formats will not be processed properly.

3.2.2. Escape Characters

3.2.2.1. General statement escape character

Every valid MSP statement is preceded by a '@' character. The statement escape character should be followed by a macro (see 3.2.2.2), constant (see 3.2.2.4) or a keyword (see 3.2.2.3) escape character immediatly. If not, the '@' is pasted into the HTML file and we do not have a valid MSP statement.

Example:

<H1>@@paste('Hello World!')</H1>

contains a valid MSP statement (a keyword with an argument list).

<A HREF=”test@test.org”>mailme</A>

is not an MSP statement. It will be passed to the client unmodified.

3.2.2.2. Macro escape character

Macros are preceded by a '@' character.

The syntax definition of a macro is:

'@' <symbol_name> '(' <argument-list> ')'

No white space characters are allowed between the macro escape character and the symbol!

3.2.2.3. Keyword escape character

Keywords are also preceded by a '@' character.

Each keyword uses its own syntax definition, see 3.2.7.

No white space characters are allowed between the keyword escape character and the symbol!

3.2.2.4. Constant value macros escape character

Constant values like version numbers or system parameters can be registered by the application or system software. They can be inserted into an MSP file using the *constant value macro*.

Constant values are preceded by a '%' character.

The syntax definition of a macro is:

'%' <symbol_name>

No white space characters are allowed between the constant escape character and the symbol!

Example:

@*%system.version* in

<h2>mCAT HTTPD Version @*%system.version*<H2>

will be replaced by the the current version of the mCAT HTTPD.

3.2.3. Symbols

A valid symbol has up to 32 characters. The first character of a symbol can be an upper or lowercase latin letter. All other characters of a symbol can be either upper or lowercase latin letter, a digit (0..9) or a dot ('.').

The dot seperates a symbol into sub-symbols used to select namespaces.

3.2.4. Namespaces

All macros and constants are stored by mCAT-HTTPD using a tree structure called *namespace*. Using the namespace, macros and constants can be grouped (and sub-grouped).

Example:

the namespace *'xio'* holds all macros and constants needed to access mCAT's ExpressIo:

@@xio.in(%xio.TSM,2,0)

The ExpressIO Function IOIn() is called. The first argument is the integer constant to refer to the TSM-Bus, the second argument selects the module and the third the channel. Executing this macro, the entire macro text is replaced by the specified IO value.

Users can create own namespaces and add macros and constant values in all namespaces available.

Please note that no macro or constant can be added to the root namespace!

3.2.5. Constant strings

As you may already have noticed, strings within an macro's argument list must be delimited by the ' (APOSTROPHE-QUOTE, hex 027) character.

The strings are assumed to be of ASCII or UTF8 format. Every non ASCII character and the *'* and *%* characters must therefore be replaced by there hex representation.

Example:

> *' The %25 and the %27 are not allowed in a string'*

will be replaced by:

> *The % and the ' are not allowed in a string*

A string may include line breaks.

3.2.6. Argument Lists

3.2.6.1. Basic Concept

It is important to understand the concept of argument lists. The basic idea starts with the CGI argument list, a well formed list of pairs. Each pair consist of an *argument name* and an *argument value*. The details of internal representation can be found in 3.5.1. Internal Representation of Argument Lists.

We have to deal with three argument lists:

1. The *global argument list* is the list that holds the arguments passed to the MSP using CGI argument notation. This argument list can be modified by so called *filter macros.*
2. The *local argument list* is an auxiliary argument list used to store values or a copy of the global argument list. The *local argument list* is also called the *local variable list* (because that is what it is used for).
3. The *macro argument list* is the current argument list passed to a macro. The macro argument list can be formed by the values of other macros (macro functions and constant macros) and constant values like numbers and strings. It is enclosed by brackets.

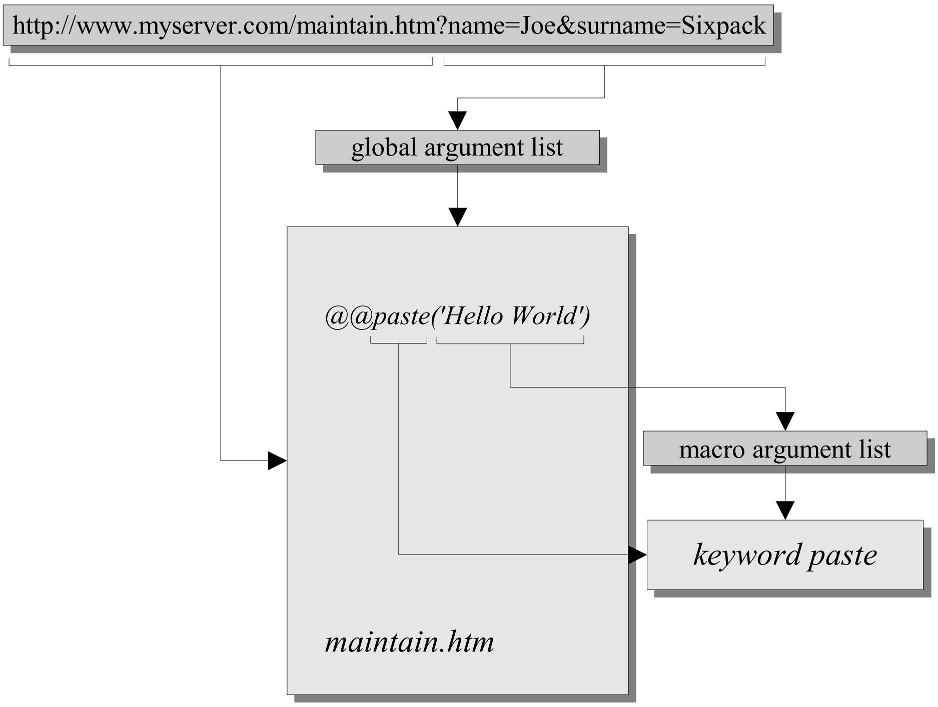


Figure 24: Global and macro argument list

There is a special character, the asterix '*', that can be used as the first argument or the entire argument list of a macro. If used, it tells a macro to use a copy of the *global argument list* as a *macro argument list*. This feature allows to call a macro using the global argument list instead of passing explicit arguments.

Note:

To understand the following example, you should know that the paste keyword evaluates the macro argument list and concatenate the results into a string that is inserted instead the macro (see 3.2.7.3. paste / tolower / toupper). You also should know that you can access arguments of the global argument list by name using square brackets around the arguments name.

Example:

Assume we have a file called *test.htm. We* request it using the following *URL:*

http://172.31.31.50/test.htm?name=Joe&surname=Sixpack

Test.htm may include:

<H1>@@paste('this',' ','is',' ','paste',' ','in',' ','action')</H1>

This evaluated to:

<H1>this is paste in action</H1>

Simple and stupid example, just to show how *paste* works. But now:

<H2>@@paste('Mr. ',[name],' ',[surname])</H2>

Is replaced by:

<H2>Mr. Joe Sixpack</H2>

And using the '*', we get:

DEBUG=*@@paste(*)*

DEBUG=JoeSixpack

Please note that no spaces or separators are inserted!!

This example may not make a lot of sense but you will see that the '*' argument is a very good helper when it comes to process more complex CGI-Requests. To give you an idea, replace the *paste* keyword in the example by some meaningful function. Lets assume you have written a function to log all the people who have done some maintenance with your installation or machine. Every technician who does maintenance has to enter a name and date of last maintenance. Lets assume you registered this function as 'maintlog.add':

The technician has to leave his name in an HTML form. When he submit this form the data is passed to the MSP file:

http://172.31.31.50/maintain.htm?name=Joe&surname=Sixpack

Your *maintain.htm* may now contain:

```
@@if (@maintlog.add(*),'OK')
        <H1>Thanks, Mr. @@paste([name],' ',[surname])</H1>
@@else
        <H1>Sorry, could not register. Try again!</H1>
```

```
@@endif
```

The syntax definition for macro argument lists is:

value *:= <macro>* | *<constant_value>* | *<string_const>* | *<number>*

argument *:= <value>*

first_arg *:= '*'* | *<argument>*

argument_list *:= '(' {<first_arg> {',' <argument>}} ')'*

3.2.6.2. Accessing Global Arguments

There are a few special sequences to access properties from the global argument list when creating a macro's argument list:

- [?] is replaced by the number of arguments in the list
- [<name>] is replaced by the value of argument <name> of the list
- [<index>] is replaced by the value of argument at position <index>

3.2.6.3. Accessing Local Variables

There are a few special sequences to access properties from the local variable list when creating a macro's argument list

- {?} is replace by the number of arguments in the list
- {<name>} is replaced by the value of the argument <name> of the list
- {<index>} is replaced by the value of argument at position <index>

3.2.7. Keywords

3.2.7.1. Local / Argv

The keyword ***local*** is used to add a local variable to the local variable list. To refer to a local variable, see 3.2.6.3. Accessing Local Variables. The keyword ***argv*** allows to set / override a value in the *global argument list*. The syntax is identical for both keywords.

The syntax is

```
@local(<name>,<value>)
```

Example:

@@local('myaddr',[addr])

create a local variable named 'myaddr' and assigns it the value of the argument (from the global argument list) 'addr'.

3.2.7.2. Load / Store

Using *store* it is possible to copy the *argv* argument list to *local,* the list of local variables. Please note that *local* is overwritten and all existing variables are deleted.

Using *load* it is possible to copy the list of local variables to *argv* argument list. Please note that *argv* is overwritten and all existing variables are deleted.

Example:

```
<!-- presume we are called with t=5 -->
@@store
@@paste('temperature=',{t},'°C')<br>
```

The result will be “temperature=5°C”.

3.2.7.3. paste / tolower / toupper

Those keywords paste the arguments from the macro argument list into the output. *Paste* does not modify the values, *tolower* converts all values to lower case and *toupper* converts all values to upper case.

```
      @@toupper('Hello out there')
```

will be replaced by

HELLO OUT THERE

3.2.7.4. foreach / for / endfor

There are two forms of the for statement.

The general form is

```
@@for('<varname>',<from>,<to>{,<step>})
```

varname = name of the for counter variable (created in namespace 'var).

from = start value

to = end value

step = increments per step (optional, default=1)

```
@@endfor
```

If *to* is smaller than *from*, the counter is decremented instead of incremented by *step*! If the counter variable must be refered inside the for loop, the namespace *var* must be added:

```
<!-- list all arguments -->
@@for('i',0,[?])
      @@paste([%var.i])
@@endfor
```

The less general version of *for / endfor* is *foreach*. It is preset to iterate through the arguments of the global argument list. The counter variable for a foreach / endfor loop is var.for. Foreach statements can not be nested.

```
<!-- list all arguments -->
@@foreach
      @@paste([%var.for])
@@endfor
```

Please note that both examples do the same job!

3.2.7.5. if / elseif / else / endif / not

The syntax for the ***if*** statement is pretty straight forward:

```
@@if(<arg1>,<arg2>)          or        @@if(<arg>)

     ...

@@elseif(<arg1>,<arg3>)      or        @@elseif(<arg>)

     ...

@@else

     ...

@@endif
```

If two arguments are given, they are compared literally and case sensitive. If only one argument is given, the expression is true if the argument exists and its value is neither 0 nor an empty string.

The logic value can be inverted by use of the keyword *not:*

```
@@if(@not(<arg1>,<arg2>))
```

```
        or

    @@if(@not(<arg>))
```

3.2.7.6. while / endwhile

The *while* keyword uses the same logical expression syntax *if* uses. *While / endwhile* forms a loop construct:

```
    @@while (<arg1>,<arg2>)        or        @@while(<arg>)

         ...

    @@endwhile
```

The loop is entered and executed as long as the logical expression in the while statement is true.

3.2.7.7. repeat / until

The *until* keyword uses the same logical expression syntax *if* uses. *Repeat / until* forms a loop construct:

```
    @@repeat

         ...

    @@until (<arg1>,<arg2>)        or        @@until(<arg>)
```

The loop is allways entered. It is executed until the logical expression in the until statement becomes true.

Note: The constructs for, while, repeat and if can be nested. The nesting level is limited to 8 levels.

3.2.8. Functions and Filters

The basic difference between a *function* and a *filter* is that a *function* returns a visible string that replaces the function statement. A *filter* modifies or replaces the global argument list and is removed from the source file:

@@int.add('1','2') is replaced by 3

In contrast,

@@xio.vecein(%xio.TSM,'1')

loads *global argument list* with the i/o values of module '1' / TSM-Bus.

In both cases the macro text itself is removed from the html file.

3.3. A few Considerations on HTML

Among the various HTML statements there are three meta tag statements that are very important to fine tune your MSP pages:

<META HTTP-EQUIV="cache-control" content="no-cache">

This meta tag should be included in every MSP page. It tells a browser to read the page allways from its source (our mCAT HTTPD-Server) instead of using a cached version of the page. If this tag is not used, you may see an older cached version of your MSP instead of the current version.

<META HTTP-EQUIV="Refresh" CONTENT="<*timeout*>; URL=<*url*>">

This meta tag can be used to automatically refresh the website with a fixed period. The argument <timeout> specifies the refresh cycle time in seconds. Setting <timeout> to **0**, will force an immediate reload! This statement can also be used to switch conditionally to another page using the ***if*** statement! If URL is given, the specified page is loaded. If no URL is given, the current page is reloaded.

<META name="ROBOTS" content="NOINDEX, NOFOLLOW">

It is also a good idea to prohibit search engines to index your pages (if they ever get access to the node).

3.4. Intrinsic Extensions

3.4.1. Intrinsic Namespace 'xio'

Group	***Function***	***F***	***Comment***
Exploration and Configuration	xio.getcfg(bus)	X	load the global argument vector with the ID's of the modules attached to ***bus***.
	xio.isdigital(bus,module)		True if module is a digital i/o module
	xio.isinput(bus,module)		True if module is a input module
	xio.isanalog(bus,module)		True if module is a analog i/o module
	xio.iscount(bus,module)		True if module is a event counter i/o module
	xio.ispos(bus,module)		True if module is a position i/o module
	xio.ispwm(bus,module)		True if module is a PWM i/o module
	xio.isfreq(bus,module)		True if module is a frequency counter i/o module
	xio.isfloat(bus,module)		True if module is a floatpoint variable table
	xio.isint(bus,module)		True if module is a integer variable table
	xio.info(bus,module,chan,item)		Read a information item (see ExpressIo documentation)
	xio.cfg(bus,module,chan,item,value)		Write a configuration item (see ExpressIo documentation)
Read/Write	xio.vecin(bus,module)	X	load the global argument vector with the io of ***module*** attached to ***bus***.
	xio.vecout(bus,module,..)		Output the values in the argument list to ***module*** attached to ***bus***.
	xio.in(bus,module,chan)		Read a single channel from.
	xio.out(bus,module,chan,value)		Output a single channel.
	xio.inobj(name)		Read a single IOOBJECT by Name.
	xio.outobj(name,value)		Write a single IOOBJECT by Name.
	xio.outlist(bus,module,..)		Accepts a list of the outputs to be set. Needed to process Form data (input) (bus=1,module=2,A1=1, A5=0)
	xio.outcheck(bus,module,..)		Accepts a list of the outputs to be inverted. Needed to process Form data (checkbox) (bus=1,module=2,A1=1, A5=1)

Group	***Function***	***F***	***Comment***
Miscellaneous	xio.triggerwd(on)		If *on* is true ('1'), a background process ist started to trigger all outputs to prevent watchdog action. The process can be disabled by calling this function with on = ' 0'
Constant	xio.CPU		A constant identifying the CPU bus
	xio.TSM		A constant identifying the TSM bus
	xio.I2C		A constant identifying the I^{2}C bus
	xio.ASI		A constant identifying the ASI bus
	xio.CAN		A constant identifying the CAN bus
	xio.SFT		A constant identifying the SFT (Software) bus

Table 7: HTTPD Intrinsic functions: ExpressIO

3.4.2. Intrinsic Namespace 'int'

Group	***Function***	***Replacement***	***Comment***
Arithmetic			
	int.add(a,b)	a + b	
	int.sub(a,b)	a - b	
	int.mul(a,b)	a * b	
	int.div(a,b)	a / b	
	int.mod(a,b)	a modulus b	
	int.neg(a)	- a	
Shift			
	int.shl(a,b)	a << b	
	int.shr(a,b)	a >> b	
Bitwise logic			
	int.ldbit(a,b)	(a & (1 << b)) >> b	Inserts ***a*** **1** if bit b is **1** else **0**
	int.stbit(a,b,c)	(((c & 1) << b) | (a & (1<<b))) >> b	bit **0** of ***c*** is inserted at bit position ***b*** in ***a***
	int.or(a,b)	a | b	bitwise and
	int.and(a,b)	a & b	bitwise or
	int.xor(a,b)	a ^ b	bitwise exclusive or
	int.cpl(a)	~ a	bitwise invert
Logic			
	int.gt(a,b)	if (a > b) 1 else 0	
	int.ge(a,b)	if (a >= b) 1 else 0	
	int.lt(a,b)	if (a < b) 1 else 0	
	int.le(a,b)	if (a >= b) 1 else 0	
	int.eq(a,b)	if (a = b) 1 else 0	
	int.ne(a,b)	if (a <> b) 1 else 0	
	int.not(a)	if (a = 0) 1 else 0	
	int.valid(a)	if (is_integer(a)) 1 else 0	
Constants			
	int.true	1	
	int.false	0	
Miscellaneous			
	int.tohex(a)	The hex value of 'a'	We use C-Like encoding like 0x66f6

Table 8: HTTPD Intrinsic Functions: INT

3.4.3. Intrinsic Namespace 'system'

Group	*Function*	*Replacement*	*Comment*
EEPROM Read/ Write	system.read(a)	value of cell 'a'	
	system.write(a,b)	none	Write value 'b' into cell 'a'
Miscellaneous	system.reset()	none	Schedule Reset. The reset will be executed when the current page is transfered completely to the host.
Constant	system.version	HTTPD-Server version	

Table 9: HTTPD Intrinsic Functions: SYS

3.5. Custom Extensions

The description of user extensions is related to the Example in 3.5.7. A Customer supplied MSP-Function Example.

3.5.1. Internal Representation of Argument Lists

Any argument list, no matter if it is the global, local or macro argument list, is internally represented in a C data structure called ARGV. The structure holds an integer value *argc* that holds the number of arguments in the list. The vector *argv* holds the argument pairs. Every argument consist of an *argument name* and an *argument value*. These values are C strings, no matter if they hold numbers or real strings. Please note that name and value are pointers. The values are allocated from the httpd's memory pool. If you add new arguments, you MUST use the API functions provided (see 3.5.5. The HTTDP API), because they will automatically allocate memory from the pool.

```
#define MAX_ARGC                        40

typedef struct {
    UTF8        *name;          // string, name of the argument
    UTF8        *value;         // value
} ARG;

typedef struct {
    UNSIGNED    argc;           // number of arguments in index
    ARG         retval;         // return value, not used for functions
    ARG         argv[MAX_ARGC];
} ARGV;
```

3.5.2. The Server Handle

The mCAT HTTPD servers are logically independent program units. Adding a functionality has only affect to one server. If there is more than one server, the new functionality must be added to each server individually. Most API functions require a server handle (sometimes referred to as *sid* for *server id*)

The API-Function

```
      sid = HTTPDQuery("MyWebSite");
```

returns a handle that refers to a specific server. Please note that the name used in this query is the name of the server as agreed on in *config.db – in this example the name is MyWebSite.*

If the returned sid is NULL, the server was not found!

3.5.3. Registering a Namespace

To implement new functions, filters or constant macros it is recommended to create a new *namespace* first. All created entities will be stored there. This makes it easy to prevent naming conflicts. To create a new namespace, use the call:

```
      HTTPDCreateNamespace(sid,"my");
```

In this example we create the root namespace *my.* If we register a function in this namespace later and the functions name is *power*, the function will be available from MSP as *my.power().*

3.5.4. Adding Macros

3.5.4.1. Adding Constant Macros

There are three API functions available to add constant macros:

```
INTEGER HTTPDAddNumConst (HTHANDLE sid,UTF8 *ns,UTF8 *name,INT32 value);
INTEGER HTTPDAddStringConst (HTHANDLE sid,UTF8 *ns,UTF8 *name,UTF8 *value);
INTEGER HTTPDAddFloatConst (HTHANDLE sid,UTF8 *ns,UTF8 *name,double value);
```

All those functions take the server id (sid), the namespace (ns) and the name of the new constant macro. They differ in the data type of the constant only.

3.5.4.2. Adding Function and Filter Macros

From the outside, registering a function or a filter looks very similar:

```
INTEGER AddFunction (HTHANDLE sid,UTF8 *ns,UTF8 *name,void *call,void *data,INTE-
GER argc,UTF8 **argnames);

INTEGER AddFilterFunction (HTHANDLE sid,UTF8 *ns,UTF8 *name,void *call,void
*data,INTEGER argc,UTF8 **argnames);
```

Again, the functions take the server id (sid), the namespace (ns) and the name of the new macro. The argument *call* is a pointer to the user supplied function that implements the new macro. The pointer *data* is an user supplied pointer to some private data. This pointer is passed to the implementation function for the programmers convenience. The integer *argc* tells the system how many arguments the user supplied function expects in an argument vector. Finally, *argnames* is a pointer to an array of *argc* strings providing the names of the expected arguments.

In our example, we expect one argument and we name it *a1*.

```
const char *arglist[] = {
    "a1"
};
```

We use the function HTTPDAddFunction to register the function then:

```
HTTPDAddFunction(sid,"my","power",power2,NULL,1,arglist);
```

3.5.5. Writing Function and/or Filter Macros

The major difference between a function and a filter macro is:

1. A filter macro always returns **NULL**.
2. A filter macro **may** modify one or more arguments from the incoming argument list.
3. A function macro returns a **meaningfully value** (or NULL on fail)
4. A function **never** modifies the incoming argument list.

Here we have an example for a macro function. It reads the argument “a1” by name from the argument list (you should read arguments by name where possible). Then it calculates the return value (accu = accu * accu) and returns the result by use of HTTPDAllocFormatted(). Please note that HTTPDAllocFormatted() is a local wrapper function you have to provide. You can copy this function from the example (we need the include <*stdarg.h*> to implement the function, do not forget to include it).

Why must we use HTTPDAllocFormatted()? This function helps us in two ways. First it converts the binary value into a formatted string (all *printf* typical formating options are available, including float point types). Second and even more important, this function will create the formatted string in the HTTPD servers memory pool. This is very important, because the return value must survive the functions runtime. Local variables, even of type static, can not be used for this purpose!

```
UTF8 *power2(INTEGER sid, ARGV *args, void data)
{
    long accu;

    data = data;    // not used in this example

    if (args) {
        // get argument
        accu = atol(CGIGetArgByName(args,"a1"));
        // calc power of 2
        accu = accu *accu;
        // return as a visible string
        return HTTPDAllocFormatted(sid,"%ld",accu);
    } /* endif */

    return NULL;
}
```

With filter macros, the handling is a bit more complex. Here is an example from the intrinsic library, implementing an ExpressIO function. The function returns for each module of a bus the module name or an empty string in a way that the index into the array is equal to the modules address. See the example “5-MSP” for details.

```
PRIVATE UTF8 *xiogetcfg(INTEGER sid, ARGV *args, void *data)
{
    INTEGER i,              // some helper
            bus;            // the selected bus
    BINFO   businfo;        // a BINFO structure of ExpressIO
    UTF8    *value;         // a string helper

    // fetch “bus” from the argument list
    bus = atol(CGIGetArgByName(args,"bus"));

    // Clear argument list
    CGIClearArgList(args);

    // get businfo structure
    if (DrvGetBusInfo(bus,&businfo,sizeof(BINFO))) {
        // bus is invalid form here on!!
        // scan all possible modules of the given bus
        for (i=0;i<businfo.slots;i++) {
            // read module name
            value = (char *)IOInfo(bus,i,0,INFO_GET_IDENT_STRING,0);
            if (value == NULL) {
                // if nothing found, use a empty string
                value = "";
            } /* endif */
            // add to argument list. Value is empty
            // for non-existing modules.
            CGIAppendArg(sid,args,businfo.name,value);
        } /* endfor */
    } /* endif */

    return NULL;  // FILTER RETURNS EMPTY STRING!!
}
```

There are several ways to implement macros, but whatever your approach is, some care has to be taken:

1. Never try to override an argument using functions like strcpy! The names and values sould be assumed to be read only. Change them using the CGI-Api functions only.
2. Never return a pointer to static memory holding a dynamic (calculated) result! You can return constant string value (“This is a true constant string value”) or you have to use HTTPDAllocFormatted() to create a valid string.
3. Never try to return a local variable.
4. Never use ThreadSleep / ThreadDelay or a long running loop in an macro function, because while the function is executed, the related HTTPD server is blocked!

3.5.6. A Customer supplied MSP-Function Example – The complete Source

This example can be found in *<mcatdir>/cc/httpd/6-msp*.

```
/*
 *    classic
 *
 *    (c) 2003 mocom software GmbH & Co KG
 *
 *    File: mspfunc.c
 *
 *    Created: 05.12.2003  12:15  VG
 *    -----------------------------------------------------------------------
 *
 *    History:
 *
 *     date     time       author       comment
 *    -----------------------------------------------------------------------
 *     05.12.2003  12:15   VG
 *    -----------------------------------------------------------------------
 */
```

```
#include <mcat.h>
#include <simpleio.h>
#include <httpdlib.h>
#include <stdarg.h>

// NOTE THAT THIS IS AN INIT, NO TASK!

// A wrapper to the HTTPDAllocFormattedV function.
INTEGER HTTPDAllocFormatted(INTEGER sid, char *fmt, ...)
{
    va_list  argp;
    va_start( argp,fmt );

    return HTTPDAllocFormattedV(sid,fmt,argp);
}

// the user supplied function calculates the power
// of a value.
UTF8 *power2(INTEGER sid, ARGV *args)
{
    long accu;

    if (args) {
        // get argument
        accu = atol(CGIGetArgByName(args,"a1"));
        // calc power of 2
        accu = accu *accu;
        // return as a visible string
        return HTTPDAllocFormatted(sid,"%ld",accu);
    } /* endif */

    return NULL;
}

const char *arglist[] = {
    "a1"
};

INTEGER TaskInit (IMD *imd)
{
    int sid;

    // find the website you need.
    // if you want to register for more than one websites
    // you must repeat the registration for every single website.
    sid = HTTPDQuery("MyWebSite");
    if (sid) {
        // we found it, now create a namespace "my"
        HTTPDCreateNamespace(sid,"my");
        //in the namespace, register the function "power"
        HTTPDAddFunction(sid,"my","power",power2,NULL,1,argument list);
    } else {
        kprintf("FAIL (%d)\n",sid);
    } /* endif */

    return -1;
}
```

3.5.7. The Argument List Handling API

```
    INTEGER CGIGetIndexByName (ARGV *arglist,UTF8 *name);
```

Returns the index of the argument named *name* into the argument vector (see 3.5.1. Internal Representation of Argument Lists). If it is not found in the argument list, -1 is returned.

```
    UTF8 *CGIGetArgByName (ARGV *arglist,UTF8 *name);
```

Fetch a value by name.

```
    UTF8 *CGIGetArgByIndex (ARGV *arglist,INTEGER index);
```

Fetch an argument by its index.

```
    INTEGER CGIAddArgByName (HTHANDLE sid,ARGV *arglist,UTF8 *name,UTF8 *value);
```

Add an argument by name. If the name does not exist in the given argument list, the name/value pair is appended. If it exists, *value* replaces the value of the existing argument. Returns FALSE on fail.

```
    INTEGER CGIAddArgByIndex (HTHANDLE sid,ARGV *arglist,INTEGER index,UTF8
*name,UTF8 *value);
```

Add an argument by index. No name checking is done. If index is out of range (less than 0 or greater than *argc* + 1), the functions fails and returns FALSE.

```
    INTEGER CGIAppendArg (HTHANDLE sid,ARGV *arglist,UTF8 *name,UTF8 *value);
```

Appends an argument to the end of the argument list. Returns FALSE on fail.

```
    void CGIClearArgList (ARGV *arglist);
```

Clear all the argument vector and set *argc* = 0.

```
    INTEGER CGIPresetArgList (ARGV *arglist,INTEGER argc,UTF8 **argnames);
```

Presets an argument list a given table of names. If you do so and add names later by use of *CGIAddArgByName()* the arguments will be in exactly the same order as they are in the table *argnames*. Please do not forget to set *argc* exactly to the number of strings in *argnames*.

4. The mCAT-HTTPD-Server traditional CGI-Processing

This example can be found in *<mcatdir>/cc/httpd/2-classic*. You can use the CGI-Api functions to fetch the arguments. Also use CGIPrint() to insert HTML-Code into the output buffer. The buffer is send after its execution is complete.

```
/*
 *    classic
 *
 *    (c) 2003 mocom software GmbH & Co KG
 *
 *    File: classic.c
 *
 *    Created: 04.12.2003  11:53  VG
 *    ----------------------------------------------------------------------
 *
 *    History:
 *
 *     date       time       author   comment
 *    ----------------------------------------------------------------------
 *     04.12.2003  11:53      VG
 *    ----------------------------------------------------------------------
 */
#include <mcat.h>
#include <simpleio.h>
#include <httpdlib.h>
#include <stdarg.h>

INTEGER CGIPrint(INTEGER sid, char *fmt, ...)
{
    INTEGER len;

    va_list   argp;
    va_start( argp,fmt );

    len = CGIVPrint(sid,fmt,argp);

    va_end( argp );

    return len;
}

UTF8 *my_cgi_function(INTEGER sid, ARGV *args)
{
    UTF8    *hallo = NULL;

    if (args) {
        hallo = CGIGetArgByName(args,"what");
    } /* endif */
    if (hallo == NULL) {
        hallo = "ERROR IN CGI 'my_cgi_function'\n";
    } /* endif */

    CGIPrint(sid,"<html>\n");
    CGIPrint(sid,"  <head>\n");
    CGIPrint(sid,"    <title>mCAT ExpressIO WEB interface</title>\n");
    CGIPrint(sid,"    <META HTTP-EQUIV=\"reply-to\" content=\"info@elzet80.de\">\n");
    CGIPrint(sid,"    <META name=\"ROBOTS\" content=\"NOINDEX, NOFOLLOW\">\n");
    CGIPrint(sid,"  </head>\n");
    CGIPrint(sid,"  <body bgcolor=\"#FFFFFF\">\n");
    CGIPrint(sid,"    <table border=0 cellspacing=5 cellpadding=5>\n");
    CGIPrint(sid,"      <tr>\n");
    CGIPrint(sid,"        <td width=\"100\">\n");
    CGIPrint(sid,"          <IMG ALT=\"ELZET80\" SRC=\"/img/elzetlogo.gif\"
border=\"0\">\n");
    CGIPrint(sid,"        </td>\n");
    CGIPrint(sid,"        <td width=\"600\">\n");
    CGIPrint(sid,"           <h1>Dieses CGI sagt '%s'</h1>\n",hallo);
    CGIPrint(sid,"        </td>\n");
    CGIPrint(sid,"      <tr>\n");
    CGIPrint(sid,"    </table>\n");
    CGIPrint(sid,"  </body>\n");
    CGIPrint(sid,"</html>\n\n");

    return NULL;
}

const char *arglist[] = {
    "what"
};

INTEGER TaskInit (IMD *imd)
{
    int sid;

    sid = HTTPDQuery("VIEW");
    if (sid) {
        HTTPDAddFunction(sid,"cgi","classic",my_cgi_function,NULL,1,argument list);
    } else {
        kprintf("FAIL (%d)\n",sid);
    } /* endif */

    return -1;
}
```

VIII. MCAT Serial Driver

1. Introduction

The serial driver for mCAT V2 is much the same as the one some customers are already using with mCAT V1. However, the mCAT V2 version was enhanced carefully to support both a higher usability and a higher system throughput.

The most powerful new feature is the *ComSetHdl* function allowing a programmer to supply his own receive code without having the need to completely rewrite the serial stuff. However, the quite powerful message handler included with the previous versions of SerDrv is still available and will be used as a default handler.

SerDrv will now read the EEPROM to get default baudrate, selected parity and handshake mode, but the calls concerning these functions are still available (*ComSetSpeed*, *ComSetHs*, *ComSetMode*).

2. Basic Operation

The serial line driver »SerDrv« has two different programmers interfaces:

- A Shared Library API used to configure the driver
- A message based interface to operate the driver

2.1. Configuration

The driver can be configured by setting up the EEPROM and / or API calls, whichever is more suitable.

2.1.1. EEPROM Configuration

The EEPROM words 5 and 6 are used to configure serial port 0 (COM1) and words 7 and 8 are used for port 1 (COM2). We will describe only words 5 and 6 as 7 and 8 are similar.

EEPROM location	***SerDrv***	***Comment***
WORD 5, LSB	Baudrate	Baudrate for channel 0
WORD 5, MSB, LSN	Handshake	Handshake (NO, RTS/CTS, XON/XOFF, HALFDUPLEX) for channel 0
WORD 5, MSB, MSN	Interface	Interface (RS232, RS485 (NET-A7 SER1 only))
WORD 6, LSB	Bits per Char	Bits per Character (7 or 8) for channel 0
WORD 6, MSB	Parity	Parity (ODD,EVEN, NONE, MARK, SPACE) for channel 0

EEPROM location	***SerDrv***	***Comment***
WORD 7, LSB	Baudrate	Baudrate for channel 0
WORD 7, MSB, LSN	Handshake	Handshake (NO, RTS/CTS, XON/XOFF, HALFDUPLEX) for channel 1
WORD 7, MSB, MSN	Interface	Interface (RS232, RS485 (NET-A7 SER1 only))
WORD 8, LSB	Bits per Char	Bits per Character (7 or 8) for channel 0
WORD 8, MSB	Parity	Parity (ODD,EVEN, NONE, MARK, SPACE) for channel 1

Baudrate [bit/s]	***SerDrv [constant values]***
1200	comBR1200
2400	comBR2400
4800	comBR4800
9600	comBR9600
19200	comBR19200
38400	comBR38400
52600	comBR57600
72800	comBR72800

Handshake	***SerDrv [constant values]***	***Comment***
NO	comNOHS	No handshake at all
RTS/CTS	comRTS	Receiver signals *request-to-send* using the RTS line.
XON/XOFF	comXON	Receiver signals handshake states „ON“ + „OFF“ by use of the ASCII charachters XON + XOFF
HALFDUPLEX	comHalf	Halfduplex only valid for NET-A7 using the RS485 interface

Interface	***SerDrv [constant values]***	***Comment***
RS232	com232	Set RS232
RS485	com485	Set RS485, this is only valid for SER1 on NET-A7 (make sure to also configure your hardware correctly to use RS485)

Bits per Char	***SerDrv [constant values]***
7	com7BPC
8	com8BPC

Parity	***SerDrv [constant values]***
NO	comNONE
EVEN	comEVEN
ODD	comODD
MARK	comMARK
SPACE	comSPACE

2.1.2. API based Configuration

There is always the possibility to change the configuration using API calls. This may be useful if:

- You wish to increase / decrease the baudrate on the fly (as some serial protocols do)
- You wish to insure that no one can change parameters accidentally

Please refer to the function reference to get more detailed information.

2.1.3. Configuring »SimpleHandler«

mCAT uses a message based approach to maintain inter-task communication. The serial line driver supports message passing allowing a task to wait for serial events without consuming (valuable) CPU time.

To send something is easy. A block of data is copied into a *ComWriteMsg* message structure and sent to the driver. The data will be sent according to the baudrate, mode and handshake the driver is configured for.

Receiving is a little bit more complex. An empty message is sent to the receiver and the receiver will fill this message. However, there are some questions to answer:

At what point can a message be said to be a complete message?

How can a »start of message« be found?

How many bytes are allowed to be filled into a buffer?

To cover the more common solution to these problems an internal receive handler called the *SimpleHandler* is provided. It can be configured to:

Wait for a synchrony character to come (start of message, the sync char will not be included into the messages data field)

Close the message and send it back if one of the two terminal characters occurs (end of message)

Close the message and send it back if either a character time-out or a message time-out occurs.

And, finally, close a message and send it back if the globally (*ComSetLen*) preset message length or the current message length is reached

Please refer to the function reference for more information.

2.2. Operation

To operate the serial lines, it is necessary to set up the driver, start operation and maintain the message traffic.

2.2.1. Message Formats

The definitions can be found in the C-header »ser.h«. The constant BUFSIZ is set to 256. To change it, simply define it before including »ser.h«.

```
#define BUFSIZ      512
#include <ser.h>

typedef struct {
      MSG    msg;                /* standard mCAT message header */
      short  len;                /* length of the data field */
      char   body[BUFSIZ];       /* data field */
} ComWriteMsg;

typedef struct {
      MSG    msg;                /* standard mCAT message header */
      short  limit;              /* maximum length of body */
      short  len;                /* number of received characters */
      char   body[BUFSIZ];       /* data field */
} ComReadMsg;
```

2.2.2. Requests/Replys

An application wanting to use the serial driver will always send requests to the driver and the driver will always send replies.

2.2.3. User Supplied Rx Handler

If »SimpleHandler« does not match the applications demands, a user written handler is supported! Using the function *ComSetHdl* will attach a receive character handler to a line. This code will be called every time a receive interrupt occurres. The user supplied handler can be written using:

Assembly language

Toshiba C V1.03

Toshiba C V4.xx (__cdecl calling convention)

Toshiba C V4.xx (__adecl calling convention)

The users function can decide individually which character should be included into the buffer and which not. It can calculate and check a checksum. In short, it can do almost anything necessary to maintain an ISO layer-2. However, a user should be careful: The handler will be called from within an interrupt routine. The code should be short, fast and should not try to share variables with other functions.

2.2.4. Buffer Usage

To improve the performance of the serial line driver it is recommended to send a threshold of at least 2 messages to the receiver. This will make a new buffer available for the receiver as soon as the previous one was passed to the application! With higher baudrates and short messages more buffers are even better. However, the driver will maintain a small fifo buffer to save characters received while it has no buffer to place them in.

3. Function Reference

The first parameter of all functions is a *com channel selector*. It must be either 1 for SER0 or 2 for SER1. Future members of the Toshiba TLCS900 family may have more serial lines and will be supported as necessary.

3.1. Basic Functions

ComSetSpeed(short com, short speed);

Function: *Set the baudrate for a given channel. The EEPROM will not be changed.*

Parameter: ***com 1=SER0, 2=SER1***

speed = comBR300 ... comBR76800

Please note that 300 baud is not available on the ELZET80 TLCS900H core CPU's (BIT900, TSM900, NET900H, NET900H+)

Return: ***-/-***

ComSetMode(short com, short mode, short parity);

Function: ***Set the mode and parity for a given channel. The EEPROM will not be changed.***

Parameter: ***com 1=SER0, 2=SER1***

mode = com7BPC, com8BPC, com9MARK, com9SPACE

parity = comNONE, comODD, comEVEN

Return: ***-/-***

ComSetHs(short com, short handshake);

Function: *Set the handshake protocol for a given channel. The EEPROM will not be changed.*

Parameter: ***com 1=SER0, 2=SER1***

handshake = comNOHS, comRTS, comXON, comHALF

If RTS/CTS handshake is used ComSetTimeout should be used with a message ***timeout of 255. Please refer to ComSetTimeout.***

comHALF can only be used with com485 on NET-A7 SER1.

Return: ***-/-***

ComSetInterface(short com, short interface);

Function: *Set the interface for a given channel. The EEPROM will not be changed.*

Parameter: ***com 1=SER0, 2=SER1***

handshake = com232, com485

*Be aware: com485 is only valid on NET-A7 SER1. It must be used with Handshake set to comHALF****. Make sure that your hardware is configured correctly to use com485.***

Return: ***-/-***

ComOperation(short com, short on);

Function: ***Enable operation for a given channel. Interrupts will be attached and operation will start.***

Parameter: ***com 1=SER0, 2=SER1***

on = TRUE;

Return: ***-/-***

3.2. SimpleHandler

There are some functions to configure the »SimpleHandler« if needed. A user supplied handler can be attached using the *ComSetHdl* call. By default the driver will try to fill the message until it is full.

ComSetSyncChar(short com, short char);

Function: ***Set the synchronization character.***

Parameter: ***com 1=SER0, 2=SER1***

char = 0 ***no sync char***

char = comTAG + 'X' ***Set sync char to »X«***

Return: ***-/-***

ComSetEnd1Char(short com, short char);

Function: ***Set first of two terminal characters.***

Parameter: ***com 1=SER0, 2=SER1***

char = 0 ***no terminal char 1***

char = comTAG + 'X' ***set terminal char 1 to »X«***

Return: ***-/-***

ComSetEnd2Char(short com, short char);

Function: ***Set second of two terminal characters.***

Parameter: ***com 1=SER0, 2=SER1***

char = 0 ***no terminal char 2***

char = comTAG + 'X' ***set terminal char 2 to »X«***

Return: ***-/-***

ComSetAddChar(short com, short char);

Function: ***Set the number of characters to include in the message after a terminating character has been received.***

Parameter: ***com 1=SER0, 2=SER1***

char = n ***number of characters to add to message***

Return: ***-/-***

ComSetLenLimit(short com, short len);

Function: ***Set a global msg length limit. If this limit is reached, the message is send back to the application.***

Parameter: ***com 1=SER0, 2=SER1***

len = 0..32768

Return: ***-/-***

*ComSetCharSet(short com, void *set);*

Function: ***Tell the driver only to accept char that are member of a set.***

Parameter: ***com 1=SER0, 2=SER1***

set = pointer to a bitmap (256 bits = 8*32 byte). For every valid char there should be the corresponding bit set.

Return: ***-/-***

ComSetTimeouts(short com, short char_timeout, short msg_timeout);

Function: ***Set the time-outs.***

Parameter: ***com 1=SER0, 2=SER1***

char_timeout = the maximum time in units of 10ms that is allowed between two consecutive characters.

msg_timeout = the maximum time in units of 10ms that is allowed for a entire message to be received.

For both parameters the valid range is 0..255. 0 will disable timeout checking, 255 will disable too, but keep the timing unit running. This option can (and should) be used to guard the RTS/CTS handshake!

Return: ***-/-***

*ComSetHdl(short com, short (*hdl)(ComReadMsg *msg, char c, char err);*

Function: ***Attach a new, user supplied RX-character handler.***

Parameter: ***com 1=SER0, 2=SER1***

hdl = pointer to the users function

Return: ***-/-***

The users function must only return one of two valid return codes:

comWaitNext:	***Character read and no action. Waiting for the next char.***
comReady:	***Message complete, please reply to application.***

All other return values will be interpreted as error code and the message will be send back! The return code will be reported in ComReadMsg.msg.error in those cases.

3.3. Auxilary Functions

ComRxFlush (short com);

Function: ***Clear the RX fifo and the current message.***

Parameter: ***com 1=SER0, 2=SER1***

Return: ***-/-***

ComTxFlush (short com);

Function: ***Clear the current TX message and send it back to application.***

Parameter: ***com 1=SER0, 2=SER1***

Return: ***-/-***

3.4. Modem Line Handling

ComSetDTR (short com);

Function: ***Set the DTR handshake line. Can be used to control a modem!***

Parameter: ***com 1=SER0, 2=SER1***

Return: ***-/-***

ComResDTR (short com);

Function: ***Reset the DTR handshake line. Can be used to control a modem! The modem will disconnect if DTR is dropped.***

Parameter: ***com 1=SER0, 2=SER1***

Return: ***-/-***

ComGetDCD (short com);

Function: ***Get the state of the DCD handshake line. Can be used to control a modem! DCD will be activ if the attached modem detects a carrier (= it is online).***

Parameter: ***com 1=SER0, 2=SER1***

Return: ***TRUE if DCD is activ!***

4. The SimpleIO Functions

SimpleIO is a streaming interface designed for debugging purposes. It is not very powerful and not recommended to be used for application design purposes.

To use this functions, you have to include the header file *simpelio.h.*

Note: With mCAT 2.20, you can use the standard C printf, fprintf functions as well to send debug data to the serial line. However, you MUST NOT INCLUDE the C *stdio.h* include file to use them! Use *simpelio.h only*! Please note that floating point types (*double* and *float*) are not supported. To output those, use sprintf to format a buffer and send the buffer to the serial line using printf.

With the SIOxxxxxx functions you can address all UARTS in a system by means of argument *channel*. If *channel* is -1, the output is send to the default UART. The default UART is the one used by SYSMON.

Function	***Comment***
SIOWrLn(channel);	Generates line break on port
SIOWrDecShort(channel,value,lead);	Writes value as short decimal *value* at port. If *lead* is true, leading zeros will be generated.
SIOWrDecLong(channel,value,lead);	Writes value as long decimal *value* at port. If *lead* is true, leading zeros will be generated.
SIOWrDecWord(channel,value,lead);	Writes value as unsigned short decimal *value* at port. If *lead* is true, leading zeros will be generated.
SIOWrDecLWord(channel,value,lead);	Writes value as unsigned long decimal *value* at port. If *lead* is true, leading zeros will be generated.
SIOWrHexByte(chan,value)	Writes value as a single byte hex value at port.
SIOWrHexWord(chan,value)	Writes value as short hex value at port.
SIOWrHexLWord(chan,value)	Writes value as long hex value at port.
SIOWrStr(chan,string)	Send string at port.
SIOWrChar(chan,chr)	Send single char *chr at* port.

Function	***Comment***
SIORdChar(chan)	Read one char from the port and return it. The calling Task/Thread is blocked while waiting.
SIOkbhit(chan)	Test if a char is available. Returns TRUE if so. Function does not block. To prevent blocking, use SIOkbhit() to determine whether SIORdChar can be called without block.
SIODumpHex(chan,addr,data)	Writes the memory dump with a length of 16 Bytes like: Fictitious adress, Data(hex), Data(hex), (16x)..., ASCII representation (00800000 AA 55 89 00 80 00 D0 00 80 00 00 04 00 00 00 00 * .U.............) to port.

4.1. SimpleIO Function directed to the default UART (SYSMON)

There is also a set of macro functions available that implicitly use -1 for the channel selector:

Function	***Comment***
WrLn();	Generates line break at port
WrDecShort(value);	Writes value as short decimal *value* at port.
WrDecLong(value);	Writes value as long decimal *value* at port.
WrDecWord(value);	Writes value as unsigned short decimal *value* at port.
WrDecLWord(value);	Writes value as unsigned long decimal *value* at port.
WrHexByte(value)	Writes value as a single byte hex value at port.
WrHexWord(value)	Writes value as short hex value at port.
WrHexLWord(value)	Writes value as long hex value at port.
WrStr(string)	Send string at port.
WrChar(chr)	Send single char *chr at* port.

Function	***Comment***
RdChar()	Read one char from the port and return it. The calling Task/Thread is blocked while waiting.
kbhit()	Test if a char is available. Returns TRUE if so. Function does not block. To prevent blocking, use SIOkbhit() to determine whether SIORdChar can be called without block.
DumpHex(addr,data)	Writes the memory dump with a length of 16 Bytes like: Fictitious adress, Data(hex), Data(hex), (16x)..., ASCII representation (00800000 AA 55 89 00 80 00 D0 00 80 00 00 04 00 00 00 00 * .U.............) to port.

4.2. Disabling SYSMON (mCAT2.20)

If you want to read a character from the default UART, you compete with SYSMON for the next char. That will not work!

So if you need to disable SYSMON for some reasons, use the new system function ***SysmonEnable(FALSE)*** to disable SYSMON. However, you should not forget to re-enable SYSMON after all by calling ***SysmonEnable(TRUE).***

SysmonEnable() returns a boolean value. If TRUE operation was successful, FALSE if not.

IX. mCAT Date and Time Library

1. Introduction

1.1. Data Types

All DateTime specific data types, structures and constants are defined in "`datetime.h`". It is located in the "`cc\include`" folder of your mCAT-distribution. Include "`datetime.h`" in your C-source file if you plan to use the Date and Time Services of mCAT.

1.1.1. The MTIME Data Type

MTIME is the data type used by the MTime-Api. It always represents a number of seconds. This can be the number of seconds since 1970-01-01 00:00:00, known as UNIX-time, or the number of seconds since system startup.

```
typedef unsigned long MTIME;   /* UnixTime,secs since 1970-01-01 00:00:00*/
```

1.1.2. The SYSTIME Structure

The SYSTIME data structure contains the broken down Gregorian date value. It is defined as follows:

```
typedef struct {
    short   millisecond;       /* millisecs after the second -- [0, 999] */
    short   second;            /* seconds after the minute   -- [0, 60]  */
    short   minute;            /* minutes after the hour     -- [0, 59]  */
    short   hour;              /* hours since midnight       -- [0, 23]  */
    short   day;               /* day of the month           -- [1, 31]  */
    short   month;             /* months since January       -- [0, 11]  */
    short   year;              /* years since 1900                       */
    short   wday;              /* days since Sunday          -- [0,  6]  */
} SYSTIME;
```

1.1.3. The TIMEZONE Structure

The TIMEZONE data structure is defined as follows:

```
typedef struct {
    char        std_name[12];       /* name string, GMT, CET, WET,...          */
    SW_TIME     std_start;          /* when to switch to standard time(in DST) */
    long        std_offset;         /* offset to UTC, local=UTC+std_offset(sec)*/
    char        dst_name[12];       /* name string, BST, CEST, WEST,...        */
    SW_TIME     dst_start;          /* when to switch to DST (in STD time)     */
    long        dst_offset;         /* offset to std, dlt=local+dst_offset(sec)*/
      long         dst_active;
} TIMEZONE;
```

1.1.4. Daylight Saving Time Switching Times

```
typedef struct {
    short month;                    /* month (Jan is 0)                  */
    short day;                      /* 1...5                             */
    short wday;                     /* weekday (Sun is 0)...             */
    short hour;                     /* hour in local time (std or dst)   */
} SW_TIME;
```

1.1.5. Predefined Daylight Saving Times

There are two predefined Daylight Saving Time settings which can be stored by enumeration in EEPROM at the moment (They are compatible to the EEPROM usage of emBASIC for timezones).

```
/* There are some predefined DST Rules. Use them with the EEPROM functions. */
#define      IDST_NONE    0
#define      IDST_EUROPE  1            /* European                          */
#define      IDST_USA     2            /* USA                               */
```

The predefined European daylight saving time uses this settings:

Offset to standard time	3600sec	
Start	`SW_TIME = {2,5,0,2}`	Start at last Sunday in March at 02:00 local time.
Stop	`SW_TIME = {9,5,0,3}`	Stop at last Sunday in October at 03:00 local dst time.

The predefined USA daylight saving time uses this settings:

Offset to standard time	3600sec	
Start	`SW_TIME = {3,1,0,2}`	Start at first Sunday in April at 02:00 local time.
Stop	`SW_TIME = {9,5,0,2}`	Stop at last Sunday in October at 02:00 local dst time.

1.1.6. MSGID

Use this constant definitions to query the MsgIds of the schedule messages:

```
#define MSGNAME_SCHEDULEAT       "mCAT/Schedule/At"
#define MSGNAME_SCHEDULEEVERY    "mCAT/Schedule/Every"
```

1.1.7. Errors

Every function of the DateTime library returns information about its exit status. There are two kinds of functions, those which return an error code, and those which return a value. The error codes are defined in "datetime.h" and printed below. If a function returns a value (usually that is a MTIME) a return value of (-1) indicates that an error occured. The defined error values are:

```
#define DATETIME_ERR_OK                0    /* no error                         */
#define DATETIME_ERR_INTERNAL          1    /* library error                    */
#define DATETIME_ERR_PARAM             2    /* parameter is not correct         */
#define DATETIME_ERR_CONV              3    /* error converting time            */
#define DATETIME_ERR_RTC               4    /* error accessing RTC              */
#define DATETIME_ERR_NOTIMPLEMENTED  5    /* this function is not implemented*/
#define DATETIME_ERR_SCHEDULELOST      11   /* the system wasn't able to send
                                               a schedule event at the correct
                                               time                             */
```

2. Mtime

2.1. Function Reference

MTimeSet

Function: Set the clock by an UNIX-style time value.

The second counter of the internal clock is set by the parameter value mTime, the millisecond counter is set to zero. The system startup time is set to the new time. mTime contains the new time in seconds since 1970-01-01 00:00:00. The time base is UTC.

The new time value is also stored in the RTC, if there is one.

C-Prototype: **short MTimeSet(MTIME mTime);**

Arguments: MTIME mTime New UTC time value in seconds since 1970-01-01 00:00:00 ("Unix-time").

Returns: Error code
DATETIME_ERR_OK
DATETIME_ERR_INTERNAL
DATETIME_ERR_CONV
DATETIME_ERR_RTC

MTimeGet

Function:	Get the clock value as UNIX-style time value. Return the actual number of seconds since 1970-01-01 00:00:00. Time base is UTC. The millisecond counter of the internal clock is ignored. If there is no RTC and you never set the time before (by calling MTimeSet() or SysTimeSet()), the number of seconds since system startup is returned.	
C-Prototype:	**MTime MTimeGet();**	
Arguments:	None	
Returns:	MTime	Actual UTC time value in seconds since 1970-01-01 00:00:00 or seconds since system startup. (-1) if an error occured

MTimeSinceStart

Function:	Get the uptime. Gets the time in seconds since the system started or since the last call to MTimeSet() or SystemTimeSet().	
C-Prototype:	**MTIME MTimeSinceStart();**	
Arguments:	None	
Returns:	MTime	Uptime in seconds. (-1) if an error occured

MTime2SysTime

Function:	Convert an UNIX-style time value to a broken down SYSTIME representation. The SYSTIME.millisecond data item is not set by this function.	
C-Prototype:	**short MTime2SysTime(SYSTIME *pSysTime,MTIME mTime);**	
Arguments:	SYSTIME *pSysTime	Pointer to SYSTIME structure, in which the converted time value is stored.
	MTIME mTime	MTIME value which should be converted (seconds since 1970-01-01 00:00:00).
Returns:	Error code	DATETIME_ERR_OK DATETIME_ERR_PARAM DATETIME_ERR_CONV

3. SysTime

4. Function Reference

SysTimeSet

Function:	Set the clock by a SystemTime-style time value. The second counter of the internal clock is set by the valueof pSysTime, the millisecond counter is set to zero. The system startup time is set to the new time. The time base is UTC. The new time value is also stored in the RTC, if there is one.	
C-Prototype:	**short SysTimeSet(SYSTIME *pSysTime);**	
Arguments:	SYSTIME *pSysTime	Pointer to a SYSTIME structure which contains the new time.
Returns:	Error code	DATETIME_ERR_OK DATETIME_ERR_INTERNAL DATETIME_ERR_PARAM DATETIME_ERR_RTC

SysTimeGet

Function:	Get the clock as SystemTime-style value. The pSysTime structure is filled with the actual time. If there is no RTC and the time was not already set by MtimeSet() or SysTimeSet() the actual time is the time elapsed since system startup plus 1970-01-01 00:00:00.	
C-Prototype:	**short SysTimeGet(SYSTIME *pSysTime);**	
Arguments:	SYSTIME *pSysTime	Pointer to a SYSTIME structure in which the time value is stored.
Returns:	Error code	DATETIME_ERR_OK DATETIME_ERR_INTERNAL DATETIME_ERR_PARAM DATETIME_ERR_CONV

SysTime2MTime

Function:	Convert a time in SystemTime style to an UNIX-style time value. The SYSTIME.millisecond data is not used by this function.	
C-Prototype:	**MTIME SysTime2MTime(SYSTIME *pSysTime);**	
Arguments:	SYSTIME *pSysTime	Pointer to a SYSTIME structure, which contains the time to convert.
Returns:	MTIME	Converted MTIME value (seconds since 1970-01-01 00:00:00). (-1) if an error occured

5. Timezone

5.1. Function Reference

TimezoneSet

Function:	This function sets the current timezone information of the system. It does not store it. It will be lost after a system reset. Use this function if you want to set a timezone, which is NOT included in the list of predefined timezones of DateTime. You have to call this function at every system startup.	
C-Prototype:	**short TimezoneSet(TIMEZONE *pTz);**	
Arguments:	TIMEZONE *pTz	Pointer to a TIMEZONE structure, which contains the timezone information.
Returns:	Error code	DATETIME_ERR_OK DATETIME_ERR_INTERNAL DATETIME_ERR_PARAM

TimezoneGet

Function:	Get the timezone information currently used by the system.	
C-Prototype:	**TIMEZONE * TimezoneGet(TIMEZONE *pTz);**	
Arguments:	TIMEZONE *pTz	Pointer to a TIMEZONE structure, which contains the timezone information.
Returns:	TIMEZONE	Pointer to TIMEZONE structure. NULL if an error occured

TimezoneSetEE

Function:	Set the timezone information of the system and write it to the EEPROM. It will be loaded and set at next system startup. Use this function if you want to set a timezone, which is included in the list of predefined timezones of DateTime.	
C-Prototype:	**short TimezoneSetEE(unsigned idst,long offset);**	
Arguments:	unsigned idst	Index value of predefined daylight saving time settings for the timezone (DST_NONE, DST_EUROPE, DST_USA,...)
	long offset	Offset value in seconds of standard time to UTC (local=UTC+std_offset)
Returns:	Error code	DATETIME_ERR_OK DATETIME_ERR_INTERNAL DATETIME_ERR_PARAM

6. LocalTime

6.1. Function Reference

LocalTimeSet

Function:	Set the system time, by using local time. Time base is local time, which is determined by the timezone information. This function modifies the RTC, if there is one.	
C-Prototype:	**short LocalTimeSet(SYSTIME *ptime);**	
Arguments:	SYSTIME *ptime	Pointer to a system time structure which contains the new local time.
Returns:	Error code	DATETIME_ERR_OK DATETIME_ERR_INTERNAL DATETIME_ERR_PARAM DATETIME_ERR_RTC

LocalTimeGet

Function:	Get the localized system time. Time base is local time, which is determined by the timezone information from EEPROM or set by a previous call to TimezoneSet().	
C-Prototype:	**short LocalTimeGet(SYSTIME *ptime);**	
Arguments:	SYSTIME *ptime	Pointer to a system time structure which will be filled with the local time.
Returns:	Error code	DATETIME_ERR_OK DATETIME_ERR_INTERNAL DATETIME_ERR_PARAM DATETIME_ERR_CONV

7. Schedule

7.1. Function Reference

ScheduleAt

Function: Request a schedule msg at one specific time.

The DateTime service sends a scheduling msg to the requesting task, when the time specified by the parameter “at” is reached. You can specify scheduling times with a precision of one second. The SYSTIME.millisecond value is ignored.

C-Prototype: **short ScheduleAt(SCHEDULEMSG *msg,SYSTIME *at,short self,short prio,short reply);**

Arguments:

SCHEDULEMSG *msg	Pointer to a SCHEDULEMSG, which is sent to the requester when the scheduling time is reached. You must never change any value of this data structure.
SYSTIME *at	Pointer to a SYSTIME structure which contains the requested scheduling time.
short self	The Id of the requesting task (as returned by TaskStartup(), normally stored in variable Self).
short prio	Priority of the message, which is used when it is sent to the requesting task (1..255, 1 is low priority).
short reply	Reply priority. This priority is used when the message is replied by the requesting task (1..255).

Returns:

Error code	DATETIME_ERR_OK DATETIME_ERR_INTERNAL DATETIME_ERR_PARAM

ScheduleEvery

Function: Request a schedule msg at specific intervals.

The DateTime service sends a scheduling msg to the requesting task, whenever a time interval specified by the parameter “every” is elapsed. You can specify scheduling every second, minute, hour, day, week, month and year. The SYSTIME.millisecond value is ignored.

You can stop scheduling by simply not sending a reply upon a scheduling request.

C-Prototype: **short ScheduleEvery(SCHEDULEMSG *msg,SYSTIME *every,short self,short prio,short reply);**

Arguments:

SCHEDULEMSG *msg	Pointer to a TSCHEDULEMSG, which is sent to the requester when the scheduling time is reached. You must never change any value of this data structure.
SYSTIME *every	Pointer to a TSYSTEMTIME structure which contains the requested scheduling time. Fill values you do not need with (-1). For example, if you want to request a scheduling event every second fill all items of this structure with (-1). If you want to schedule every minute, set the seconds when you want it to get scheduled and fill all other values with (-1).
short self	The Id of the requesting task (as returned by TaskStartup(), normally stored in variable Self).
short prio	Priority of the message, which is used when it is sent to the requesting task (1..255, 1 is low priority).
short reply	Reply priority. This priority is used when the message is replied by the requesting task (1..255).

Returns:

Error code	DATETIME_ERR_OK DATETIME_ERR_INTERNAL DATETIME_ERR_PARAM

ScheduleEvery

DATETIME_ERR_CONV

X. BgMem: Nonvolatile Data Storage for mCAT

1. Introduction

BgMem allows the storage of data in battery backed up RAM, this implies that it will only run on hardware with battery. The functions are similar to a file system, there are, however, some restrictions to optimize the behaviour for real time systems:

- No dynamic file size
- No variable record sizes

On the other hand provides BgMem possibilities a normal file system lacks:

- Stack (LIFO - Last in First out)
- FIFO (First in First out)
- Ring Buffer
- Random access

BgMem stores time and date of the file generation and of all changes to the file (requires hardware with RTC)

2. Fundamentals

2.1. Organization

A BgMem memory can have a maximum of 16 files.

A file can have 1..65535 records

Size and number of records are fixed while the file is created.

File size is limited to the amount of memory set aside for BgMem.

2.2. File Names

File names can be 32 characters long and are distinguished between lower and upper case. (Lower case file names cannot be manipulated by SYSMON)

Admissable characters are “$_-.” , “A”..“Z”, “a”..“z”, “0"..”9"

Not acceptable are “mybgmem/data”, “äste/dat”

2.3. Treatment of BgMem at System Start

At system start BgMem checks whether a BgMem memory has been defined. If it finds files, they are checked for consistency. One defective record is not regarded a fatal error - it would just be deleted.

If a directory or a file is defective, it will be removed. If all files are defective, BgMem assumes the file systems has been destroyed or it has not yet been initialised and formats it new to avoid more error messages.

2.4. Memory Management

The memory for the BgMem area can be found just below SYSTEM.HEAP and above user memory. Its size is controlled by the value in EEPROM word 15, giving the size in kBytes. BgMem stores all nonvolatile data there: files, directory and some management info.

If this area is accessed without using the BgMem functions, records will be lost, perhaps also whole files.

The highest available free memory address is returned by the “mem” command of SYSMON. There, the address given at “NVRAM START ADDR” is the lowest memory used by BgMem. User programs and data must stay below this address.

2.5. File types

2.5.1. File Pointers

Write- and read access happens by using hidden (separate) write and read pointers .

For FIFO, LIFO and RING files there’s only one set of pointers, for RANDOM files there is one set of pointers for each reference to the RANDOM file so every user has its own pointer set. They are set to ‘0' when opening the file.

2.5.2. FIFO Files

FIFO files increment the write pointer with every write access. At file end it is reset to 0. If the write pointer overtakes the - otherwise independent - read pointer, no more data will be written, the write pointer gets not incremented and the function returns the “BGM_ERR_EOF” error. The file is full.

When reading from the file, the read pointer gets incremented and is also reset to 0 at the end of the file. If the read pointer overtakes the write pointer, no more data are read and the read function returns the “BGM_ERR_EOF” error.

For technical reasons, only n-1 of n records can be used.

2.5.3. LIFO Files

The last-in-first-out structure increments the write pointer with every write. At file end no more data will be written, the write pointer will not be incremented and the function returns the “BGM_ERR_EOF” error. The file is full.

When reading from the file, the write pointer will be also used as read pointer. If it gets 0, the function returns the “BGM_ERR_EOF” error. The file is empty. If the file is not empty, the write pointer gets decremented and the record is read.

2.5.4. Ring Buffers

Ring buffers are useful to store continuous data where access to just the latest records is necessary at a time. The reading task or thread should have a higher priority than the writing one to make sure data can be read faster than written, to prevent obstipation.

Basically a ring buffer is a FIFO with a different handling of the file full situation: If the write pointer would overtake the read pointer, the file isn’t signalled to be full but instead the read pointer is incremented synchronously with the write pointer and the oldest record gets lost.

2.5.5. Random Access Files

Files with random access allow the greatest flexibility in accessing records. Contrary to classical file systems (Windows/Unix) BgMem can access records post the current file end if they are within the maximum number of records specified. If an unused (unwritten) record is read, a length of 0 is returned - it is interpreted as empty record.

The 3 functions BGMWriteRandom, BGMReadRandom and BGMClear are used only with random files. BGMWriteRandom and BGMClear set the current write pointer, BGMReadRandom the read pointer independently of each other.

While opening or creating a random file, write and read pointers are set to 0. For each open file view (BGMOpen/BGMCreate) a separate set of pointers is managed, this being the main difference to the other structures.

After the read pointer is set with one of the above functions, both read and write pointer point to the following record. Thus, BGMRead and BGMWrite or BGMVPrint can be used to sequentially read or write from the random starting point.

Example (without error checking):

```
        BGMWriteRandom(hdl,“no1“,4,3);
        BGMWrite(hdl,“no2“,4);
        BGMReadRandom(hdl,buffer,4,4);    // buffer contains „no2“
        BGMReadRandom(hdl,buffer,4,3);    // buffer contains „no1“
```

3. Setup of a BgMem file

BgMem files are set up with the SYSMON “format” command. See “BgMem extensions in SYSMON”.

3.1. Special Operating Considerations

3.1.1. Changing Heap and BgMem Sizes

Modifying the size of the SYSTEM HEAP by setting EEPROM word 10 and/or modifying the BgMem size by the "format" command of the Sysmon **results in total data loss of the BgMem memory**. You should not unnecessarily change those values.

3.1.2. Creating and Deleting Files

Deleting files requires numerous restructuring operations of the memory, possibly with moving entire files. The same applies to calling the BGMCreate with a modified record size or number. Make sure the system is in low load conditions and supply voltage is safe at the time you start these transactions.

3.1.3. Data Loss From Programming Errors

The memory area for BgMem is not protected by hardware - it can be accessed from all user programs. Hence, programming errors like pointer errors can easily destroy data within BgMem. The CRC checking employed by BgMem can only detect that such errors have occured - not prevent them.

4. Function Reference

*int BGMWrite (BGMHDL bgm, void *data, word len);*

Function: Writes a record to the file that is referenced by BGMHDL. The length of the record is limited by the value specified during file creation. The real length "len" will also be stored into the record.

This function is suitable for all types of files. The record is inserted at the current write pointer position. Write pointer movement depends on the file type.

Arguments:

bgm	"Handle", value returned in BGMOpen/BGMCreate
data	Pointer to data to be stored
len	Real length of data to be written

Returns: Number of bytes written or error code. If the value is smaller than 0, it is an error code.

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks: See 2.5 file types for the different management of LIFO, FIFO, Ring buffer and Random file

*int BGMRead (BGMHDL bgm, void *data, word len);*

Function: Reads a record from the file that is referenced by BGMHDL. The length of the record is limited by the value specified during file creation. The user is responsible for *data to point to a memory block of sufficient size.

This function is suitable for all types of files. The record is copied from the current read pointer position. Read pointer movement depends on the file type

Arguments:

bgm	"Handle", value of the file returned in BGMOpen/BGMCreate
data	Pointer to data to be stored
len	Real length of buffer for data to be read.

Returns: Number of bytes read or error code. If the value is smaller than 0, it is an error code

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks: See 2.5 file types for the different management of LIFO, FIFO, Ring buffer and Random file

*long BGMVPrint (BGMHDL bgm, const char *fmt, void *data);*

Function: Converts and formats a list of data and stores it into a record. The length of the string is checked to be smaller than the file definition value. Format information follows the C function "vprintf". A terminating 0 is always inserted, hence the strings are valid C strings.

This function is suitable for all types of files. The record is inserted at the current write pointer position. Write pointer movement depends on the file type.

Arguments:	bgm	"Handle", value returned in BGMOpen/BGMCreate
	fmt	Format string
	data	Pointer to list of arguments to be stored
Returns:		0 if ok, >0 is number of excess characters.
Supported on:	mCAT	2.10 and higher
	Hardware	All with battery backup

Remarks: A sample for the use of this function can be inspected in the implementation of BGMPrint(...) in \xampl1\gen.c

*long BGMWriteRandom (BGMHDL bgm, void *data, word len, long pos);*

Function: Writes a record to the file that is referenced by BGMHDL. The length of the record is limited by the value specified during file creation. The real length "len" will also be stored into the record.

This function is suitable for random files only. The record is inserted at the position given in pos. Write pointer increments after storage.

Arguments:

bgm	"Handle", value returned in BGMOpen/BGMCreate
data	Pointer to data to be stored
len	Real length of data to be written
pos	Position of record in file. If negative, position is calculated from the end of the file. If the specified record is not available (pos outside file start or file end), the first or last record will be written to.

Returns: Number of bytes written or error code. If the value is smaller than 0, it is an error code.

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks: Using this function on a FIFO, LIFO or Ring file results in the BGM_ERR_ILLEGAL_ACCESS error.

*long BGMReadRandom (BGMHDL bgm, void *data, word len, long pos);*

Function: Reads a record from the file that is referenced by BGMHDL. The length of the record is limited by the value specified during file creation. The user is responsible for *data to point to a memory block of sufficient size.

This function is suitable for random files only. The record is copied from the position given in pos. Read pointer increments after storage.

Arguments:

bgm	"Handle", value returned in BGMOpen/BGMCreate
data:	Pointer to data to be stored
len:	Real length of buffer for data to be read.
pos:	Position of record in file. If negative, position is calculated from the end of the file. If the specified record is not available (pos outside file start or file end), the first or last record will be read.

Returns: Number of bytes read or error code. If the value is smaller than 0, it is an error code.

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks: Using this function on a FIFO, LIFO or Ring file results in the BGM_ERR_ILLEGAL_ACCESS error.

void BGMClear (BGMHDL bgm, long pos);

Function: Clears the record at the position specified - the length of data will be set to 0

This function is suitable for random files only. The record is cleared at the position given in pos. Write pointer increments after storage.

Arguments:

bgm:	"Handle", value returned in BGMOpen/BGMCreate
pos:	Position of record in file. If negative, position is calculated from the end of the file. If the specified record is not available (pos outside file start or file end), the first or last record will be written to.

Returns: Number of bytes written or error code. If the value is smaller than 0, it is an error code.

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks: Using this function on a FIFO, LIFO or Ring file results in the BGM_ERR_ILLEGAL_ACCESS error.

long BGMGetPos (BGMHDL bgm, int read);

Function:

Arguments: bgm: "Handle", value returned in BGMOpen/BGMCreate

Returns:

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks:

long BGMSetPos (BGMHDL bgm, int read, long value);

Function:

Arguments: bgm: "Handle", value returned in BGMOpen/BGMCreate

Returns:

Supported on:

mCAT	2.10 and higher

long BGMSetPos (BGMHDL bgm, int read, long value);

Hardware	All with battery backup

Remarks:

long BGMLockRecord (BGMHDL bgm, long first, word records);

Function:	Inhibits write access of other tasks to records of a random file. Use of this function on FIFO, LIFO or ring files is ignored.	
Arguments:	bgm:	"Handle", value returned in BGMOpen/BGMCreate
	first:	Number of first record to lock
	records:	Number of records to be locked. All records can be locked, even those not yet used.
Returns:		TRUE: Lock successful, FALSE Record(s) could not be locked
Supported on:	mCAT	2.10 and higher
	Hardware	All with battery backup
Remarks:	Only one lock per thread is possible. Locks are cancelled also by BGM-Close	

void BGMUnLockRecord (BGMHDL bgm);

Function:	Cancelles a lock previously installed.	
Arguments:	bgm	"Handle", value returned in BGMOpen/BGMCreate
Returns:	mCAT	2.10 and higher
Supported on:	Hardware	All with battery backup
Remarks:		

long BGMSetAttrib (BGMHDL bgm, word attr);

Function:	Setup of file attributes. Currently, only "READ-ONLY" is implemented.	
Arguments:	bgm:	"Handle", value returned in BGMOpen/BGMCreate
	attr:	0 or BGM_ATTR_RDONLY
Returns:		Error code, BGM_ERR_OK when successful
Supported on:	mCAT	2.10 and higher
	Hardware	All with battery backup
Remarks:		

*long BGMGetAttrib (BGMHDL bgm, BGMATTRIBUTE *attrib)*

Function: Read all file attributes

Arguments:

bgm: "Handle", value returned in BGMOpen/BGMCreate

attr: Pointer to a data structure of type "BGMATTRIBUTE" (see below)

Returns: Error code, BGM_ERR_OK when successful

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks:

```
typedef struct {
      lword  created;      // time created
      lword  modified;     // time last modified
      word   records;      // number of records available
      word   size;         // size of record in bytes
      byte   attr;         // file attributes
      byte   mode;         // file mode
      char   name[BGM_NAME_LEN+1]; // name
      char   padding;      // internal use only
} BGMATTRIBUTE;
```

BGMHDL BGMReadDir (word seq, char buffer);

Function: Serves to inspect all available files

Arguments:

seq: A sequential number

buffer: Pointer to a character array (string), minimum length BGM_NAME_LEN +1

Returns:

```
BGM_ERR_OK
      File exists, file name is written to "buffer"
BGM_ERR_FILE_CORRUPTED
      File contains one or more defective records
BGM_ERR_DIR_CORRUPTED
      File exists but cannot be accessed. Gets
      deleted on next restart
BGM_ERR_FILE_NOTUSED
      File exists but is not used
BGM_ERR_EOF
      Current value of "seq" is higher than highest
      entry in file table
```

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

BGMHDL BGMReadDir (word seq, char buffer);

Remarks: Example:

```
int i = 0;
char buffer[BGM_NAME_LEN+1];
do {
        error = BGMReadDir(i++,buffer);
        if (error == BGM_ERR_OK) {
                WrStr(buffer);
                WrLn();
        } /* endif */
} while (error != BGM_ERR_EOF);
```

*BGMHDL BGMCreate (char *name, word records, word size, int mode);*

Function: Creates a file. If a file of the same name exists, it gets cleared and all data is lost.

Arguments:
name: File name, max. 32 characters. “$_-.” , “A”..“Z”, “0"..”9"
records: Number of records in the file: 0 < records < 65536
size: Size of records in bytes: 0 < records < 65536
mode: Bit array consisting of one of the file types:

```
BGM_MODE_FIFO:      File has FIFO structure
BGM_MODE_LIFO:      File has LIFO structure
BGM_MODE_RING:      File has ring structure
BGM_MODE_RANDOM:    File has random access
                    optionally logically ORed with
BGM_MODE_FIT:       Ignore ”records” argument and
use                 all remaining BgMem-memory for
                           this file
For example: BGM_MODE_FIFO | BGM_MODE_FIT
```

Returns: BGMHDL reference to the file or error code. A valid BGMHDL is always > 0

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks: A create on an existing file clears that file if records or size are different to the existing file. Otherwise the file will just be initialized and set to a new file type

Also please see remarks to BGMRemove

long BGMRemove (char *name);

Function:	Clears a file	
Arguments:	name	Name of the file to clear
		BGM_ERR_OK or error code
Returns:	mCAT	2.10 and higher
Supported on:	Hardware	All with battery backup
Remarks:	***Clearing a file is a critical transaction and should be executed with caution. See chapter***	

BGMHDL BGMOpen (char *name, int wr);

Function:	Open a file, get a reference (a handle) on it	
Arguments:	name	Name of the file
	wr	TRUE means open for read and write
Returns:		BGMHDL or error code. A valid BGMHDL is always > 0
Supported on:	mCAT	2.10 and higher
	Hardware	All with battery backup
Remarks:		

void BGMClose (BGMHDL bgm);

Function:	Close a file, return the reference to it	
Arguments:	bgm	"Handle", value returned in BGMOpen/BGMCreate
		-
Returns:	mCAT	2.10 and higher
Supported on:	Hardware	All with battery backup
Remarks:		

lword BGMInit (lword size, int fmt);

Function:	Create or initialize a file. This function is used by the SYSMON function “format”	
Arguments:	size:	`Size of the BgMem memory`

lword BGMInit (lword size, int fmt);

fmt:

`BGM_CMD_GETSIZE Just find out the size of BgMem and return`

`BGM_CMD_SETSIZE Change the size of BGMEM by writing size (in kBytes) to EEPROM word 15. Thereafter a RESET is executed automatically!`

`BGM_CMD_FORMAT Format BgMEM. This should be done always after changing the size of BgMem.`

Returns: Size of BgMem or 0 if no BgMem available

Supported on:

mCAT	2.10 and higher
Hardware	All with battery backup

Remarks:

5. Error Codes

Fehlercode	***Wert***	***Beschreibung***
BGM_ERR_OK	0	No error >= 0
BGM_ERR_EOF	-1	EOF
BGM_ERR_NOT_OPEN	-2	File not open, BGMHDL not valid
BGM_ERR_FILE_NOTUSED	-3	File does not exists but could be created. Possible return value in BGMReadDir
BGM_ERR_FILE_CORRUPTED	-4	File contains invalid record(s)
BGM_ERR_NOT_INSTALLED	-5	BgMem server not installed, not started
BGM_ERR_DIR_CORRUPTED	-6	File is corrupted, will be deleted with next RESET
BGM_ERR_RECORD_CORRUPTED	-7	Current record is corrupted
BGM_ERR_RDONLY	-8	File is write protected
BGM_ERR_LOCKED	-9	Current record locked for write access
BGM_ERR_FILE_NOT_FOUND	-10	File does not exist (BGMOpen)
BGM_ERR_ILLEGAL_ACCESS	-11	File access not possible (possibly attempt to access FIFO, LIFO or ring buffer with functions for random file)
BGM_ERR_RECORD_EMPTY	-12	currently not used
BGM_ERR_OUT_OF_MEMORY	-13	BGMCreate: File could not be installed due to low memory
BGM_ERR_ILLEGAL_POS	-14	The requested file position is not valid
BGM_ERR_INVALID_FILE_NAME	-15	The file name passed with BGMCreate was not valid.

XI. mCAT Random Access Memory Management

1. Introducing mCAT V2 Memory Management

MCAT V2.00 introduced a memory management for both system memory and buffer pool management. This document will introduce the concepts and can be usd as a function reference.

1.1. System Memory Heap Management

The system heap is a chunk of memory at top of physical ram memory. Its size in kBytes can be set by changing EEPROM word 10 (0ah). If EEPROM word (10) is invalid, a default heap size is used. The size of the default heap is set at compile time and depends on the hardware resources available. The system heap is used for:

- System data structures including THREAD and TASK descriptors
- Stacks for both threads and interrupt drivers
- Message pools (for example the BITBUS pool)

The memory system is open to introduce additional heaps. The current release of mCAT does not support them. Memory once allocated will not be freed again (with the current implementation).

The system heap is heap »0«.

1.2. Message Buffer Pool Management

A special problem within a communication orientated system like mCAT is managing *buffers*. A buffer is a dynamically allocated chunk of memory of a given, fixed length. For a specific task or interrupt driver a fixed number of buffers are made available.

The message pool API implements a buffer management that suits these demands and it is optimized for fast operation, so it can be used everywhere in the system, even within interrupt drivers (BITBUS driver for example).

Because a buffer or message pool has a fixed number of buffers with a fixed length, fragmentation will not occur.

Please note that the Message Buffer Pool Management was extended with mCAT2.10-T215. See the new functions MsgPoolCreateEx(), MsgPoolGetInfo() and MsgPoolAddNotifyHdl()

1.3. Memory Address and Pointer Calculation

Within the memory space of a CPU there is a *RAM window* defined. This is the space where physical RAM will be installed. A frequently used *RAM window* starts at 400000h and is 400000h long (4MB).

It is specified that physical RAM should be mirrored within this window. A 128k ram will be mirrored 32 times for example.

That feature (it really is not a bug) is used to make mCAT run with all ram sizes from 128k up to 4MB without recompilation. The RAM size is calculated at startup time. However, because the mCAT system heap and the mCAT internal variables are located on top of the physical ram window, the addresses will not be related to the real physical location.

Using the functions of the memory API, it is simple to recalculate the pointers for debugging and system information purposes. Under normal operation, this calculation is never necessary. If however necessary, please follow this sample code:

```
lword GetTopOfMemory()
{
        lword sizeb, top;

        // get size of RAM in byte
        sizeb = 1024 * (lword) MemGetRamSize(0);
        // get NVRAM area pointer. Is not null if BGMEM is installed
        top = (lword) MemGetNVRamInfo();
        if (top == NULL) {
            // if no BGMEM, get lowest HEAP address
            top = (lword) MemAlloc(0,0,0)
        } /* endif */
        // norm. User should never use ram >= top
        top = (top % sizeb) + 0x400000;

        return top;
}
```

> *Please note that with mCAT 2.10-T215 and higher there is no need for a recalculation, because the heap addresses are automatically mapped to the real physical address space.*

2. Datatypes

With the latest changes (port to the ARM7 platform) we had to change a few datatypes. For example, the argument heap of MemAlloc() was defined as *word* previously. Now it has the portable type HEAPHDL. Because HEAPHDL is defined as a *word* on the Toshiba platform, this is still compatible in both cases, physical calling convention and compile time type checking. The introducing of the new types make it easier to support cross platform compatibility.

3. The Memory Management API

MemAlloc

Function: Allocates a chunk of memory from the given heap. If the requested memory size can not be allocated, a NULL pointer is returned.

C-Prototype: void * MemAlloc (HEAPHDL heap, UINT32 size, INTEGER clear);

Arguments:

heap	Selects a heap. Currently only the system heap (heap=0) is supported.
size	Length of the memory chunk. If the length does not obey to the systems alignment rules, the length is increased to meet them.
clear	If TRUE, the memory will be cleared (filled with 0) before return.

Returns: Aligned pointer to a chunk of memory or NULL if fail!

Supported:

mCAT	All versions.
Hardware	All

Comments:

> *If you call MemAlloc to allocate memory from the system heap (heap=0) with a size equal to 0, the function will return the lowest address used by the system!*

MemFree

Function: Frees a chunk of memory previously allocated by MemAlloc - if the heap supports freeing of memory! The mCAT SYSTEM HEAP (0) of mCAT does not support this function. However, future versions may support it and it is always a good practice to use MemFree!

C-Prototype: void MemFree (HEAPHDL heap, void *mem);

Arguments:

heap Selects a heap. Currently only the system heap (heap=0) is supported.

mem Pointer to the memory chunk to be freed

Returns: -/-

Supported:

mCAT	All versions.
Hardware	All

Comments:

MemGetFree

Function: Retrieve the size of the memory available within this heap

C-Prototype: UINT32 MemGetFree (HEAPHDL heap);

Arguments:

heap Selects a heap. Currently only the system heap (heap=0) is supported.

Returns: Memory available within this heap

Supported:

mCAT	All versions.
Hardware	All

Comments:

MemGetRamSize

Function: Returns the physical size of the RAM in use. The returned value is in kBytes, so with an 128k RAM *MemGetRamSize* will return 128.

C-Prototype: UINT32 MemGetRamSize (word reserved);

Arguments:

reserved This argument is reserved for future extension.

Returns: Size of physical RAM in kBytes

Supported:

mCAT	All versions.
Hardware	All

Comments:

4. The Buffer Pool Manager API

MsgPoolCreate

Function: Create a message pool. The chunk of memory passed by use of parameter »mem« must be big enough to contain all buffers (len*buffers) plus some management data. Please do not try to calculate this size manually. Use the function *MsgPoolCalcSize* instead.

C-Prototype: POOLHDL MsgPoolCreate (void *mem, UINT32 len, UNSIGNED buffers);

Arguments:

mem	A pointer to a chunk of memory big enough to hold all buffers.
len	The length of chunk "mem"
buffers	Number of buffers “mem” shall be divided into.

Returns: A pool handle or 0 on fail! A pool handle is the bit-inverted ordinal number of the pool. This information may be of interest when using MsgPoolGetInfo. See below.

Supported:

mCAT	All versions.
Hardware	All

Comments:

```
A typical sequence looks like this:

      UINT32 size;
      void   *mem;
      word   my_pool

      // 20 buffers, 100 bytes each
      size =  MsgPoolCalcSize (100,20);
      // get memory from heap
      mem = MemAlloc(0,size,1);
      if (mem) {
             // create!
             my_pool =  MsgPoolCreate (mem,size,20);
      } else {
             // FAIL ...                // handle error
      } /* endif */
```

MsgPoolCalcSize

Function: Calculate the size needed to create a pool of *buffers* of useable length *len*. See *MsgPoolCreate* for details.

C-Prototype: UINT32 MsgPoolCalcSize (UNSIGNED len, UNSIGNED buffers);

Arguments:	len	The usable length of a buffer
	buffers	Number of buffers desired
Returns:		Size of pool in bytes including management data.

Supported:

mCAT	All versions.
Hardware	All

Comments:

MsgPoolAlloc

Function: Allocate a buffer from *pool*. The buffer is of the given size for that pool (see *MsgPoolCreate*). An application must not write behind the end of the buffer - this will cause unpredictable errors.

The buffer is not cleared (set to »0«) and it contains random data.

C-Prototype: byte *MsgPoolAlloc (POOLHDL pool)

Arguments:	pool	A valid pool handle
Returns:		A buffer or NULL if either an error occurred or no more buffers are available.

Supported:

mCAT	All versions.
Hardware	All

Comments:

MsgPoolFree

Function: Free a buffer previously allocated by *MsgPoolAlloc*. You need not to know the *pool* handle the buffer belongs to as this will be managed automatically!

C-Prototype: void MsgPoolFree (byte *buffer);

Arguments: buffer Pointer to the buffer to be freed.

Returns: -/-

Supported:

mCAT	All versions.
Hardware	All

Comments:

FREE CAN BE USED GLOBALLY! IF YOU HAVE THE ADDRESS OF A BUFFER, YOU CAN FREE IT WITHOUT KNOWING WHAT POOL IT BELONGS TO.

MsgPoolAvailable

Function: Tells how many buffers are available in a given pool.

C-Prototype: INTEGER MsgPoolAvailable (POOLHDL pool);

Arguments: pool A valid pool handle

Returns: Number of free buffers in *pool*

Supported:

mCAT	All versions.
Hardware	All

Comments:

MsgPoolAddNotifyHdl

Function: For every pool there can be ONE *notify* function. Whenever a buffer is freed, MsgPoolFree will call a pool's notify function first (if available) to signal that a buffer is freed. The function can be used for debugging purposes as well as for elaborated buffer handling applications.

If *notify* returns TRUE, MsgPoolFree will abort its action and presume that *notify* uses the buffer for some other purpose (so it will stay allocated!). The buffer can be freed using a call to MsgPoolFree later if needed.

Please note that while *notify* is called, all interrupts are disabled. Keep *notify* as short and fast as possible!

C-Prototype: INTEGER MsgPoolAddNotifyHdl (POOLHDL pool, void *notify, void *self);

Arguments:

pool A valid pool handle

notify A pointer to a notify function, passed as a void pointer. The prototype of Notify is:

INTEGER notify (void *self, void *buffer);

self A pointer to private data that is passed along with a call to notify for the programmers convince.

Returns: TRUE if function succeed

Supported:

mCAT	All versions newer than build T00215.
Hardware	All

Comments:

MsgPoolGetInfo

Function: Retrieve information about a given pool.

C-Prototype: INTEGER MsgPoolGetInfo (POOLHDL pool, POOLINFO *info);

Arguments:

pool	A valid pool handle
info	A Pointer to a POOLINFO structure (see below)

Returns: If TRUE, there are more pools available (useful to scan all pools, see example)

Supported:

mCAT	All versions newer than build T00215.
Hardware	All

Comments:

```
typedef struct {
    UINT32      start;          // POOL start address
    UINT32      end;            // POOL end address
    UNSIGNED    b_size;         // size of a buffer
    UNSIGNED    b_count;        // total number of buffers
    UNSIGNED    b_free;         // number of available buffers
    INTEGER     owner;          // owner (0 if pool is not used)
} POOLINFO;
```

Example – scan all pools in the system:

```
UNSIGNED scan = 0;
INTEGER  more;
POOLINFO info;
do {
      // invert scan to get a pool handle!
      more = MsgPoolGetInfo(~scan,&info);
      if (more && info.owner) {
             // you got information from pool 'scan'
             ..
      }
      scan++;
} while (more);
```

MsgPoolCreateEx

Function: This function is very similar to MsgPoolCreate. The major difference is the extra *owner* argument. The owner is the taskid or intid of the creator. The id is stored in the pools information structure and can be retrieved for debugging purposes (see SYSMON command ***pools***). Because the system cannot always automatically resolve the owner – especially not in the boot phase when TaskInit is executed – we provide MsgPoolCreateEx for this purpose.

C-Prototype: POOLHDL MsgPoolCreateEx (UINT8 *mem, UINT32 len, UNSIGNED buffers, INTEGER owner);

Arguments:

mem A pointer to a chunk of memory bug enough to hold all buffers.

len The length of chunk "mem"

buffers Number of buffers “mem” shall be divided into.

owner The taskid or intid of the owner.

Returns: A pool handle or 0 on fail! A pool handle is the bit-inverted ordinal number of the pool. This information may be of interest when using MsgPoolGetInfo. See below.

Supported:

mCAT	All versions newer than build T00215.
Hardware	All

Comments: Example (from the ethernet driver):

```
pool = MsgPoolCreateEx(poolmem,1500,MAXBUFFERS,INT_ETH_DMA_RX);
if (pool == 0) {
      TraceWriteLog("\r\n ETH ERROR: CAN'T ALLOCATE POOL”);
      return FALSE;
} /* endif */
```

XII. mCAT Non-Volatile Memory Management

1. Introducing Non-Volatile Memory Management

The NVMEM shared library is a collection of API's designed to access non-volatile memory like FLASH memory and serial EEPROM's. The lib is very fundamental for mCAT and therefore usually the first module outside the kernel that is initialized.

1.1. Serial EEPROM API

The API (see chapter 3 for details) covers functions to read and write 16-Bit words from and to serial EEPROMS via an SPI or I^{2}C bus.

On Systems with SPI-Bus, both 6 and 8 bit address mode SPI-EEPROMS are supported. With mCAT2.10R00300 and later that support SPI-EEPROMS we support any address word width (the width is probed at system start).

On Systems with I^{2}C-Bus (all recent systems), both one and two address byte EEPROMS are supported.

The size of the address is auto-detected at boottime.

1.2. FLASH Memory API

There is a FLASH-API that can be used to explore, erase and program FLASH memory. Its functionality is enhanced with mCAT2.10-T00215. The new functions give access to the internal database of supported flash memories.

1.3. I^{2}C API

Even if the I^{2}C-driver does not logically fit into the NVMEM library it was placed here because it is needed to access serial EEPROMS.

2. Accessing EEPROMS

2.1. EEPROM Address Scheme

MCAT supports the use of small serial EEPROMS to store configuration data (as network address, network parameter or others). A serial EEPROM is assumed to be organized in 16-Bit units. More than one EEPROM can be used in a system (up to 3 or 4, depending on the system configuration). Parts of those EEPROMS are used to store constant values and a software scheme is provided to write protect these specific areas. This area is called the ROM section. The ROM-Section is addressed using the highest bit in an EEPROM address value. The biggest EEPROM size supported is 8196 BYTES (4096 Words!).

The 16-Bit logical address word is formed as follows:

ROM	**RES**[4]	**CHIP**	**LINEAR ADDRESS**
15	14	12,13	0..11

The size of the ROM area is 0x40 hex and it is available on every EEPROM in the system. To address this area, the highest bit of the address must be set to 1.

Bits 12 and 13 are used to select a specific chip. On some systems, only 3 chips can be addressed because of a low level I^{2}C addressing conflict (This happens when an RTC8564 realtime clock is used).

The per chip linear address can range from 0 to 0xFFF (4096 words).

	LOGICAL ADDRESS	***LINEAR ADDRESS***	***ROMBIT***	***Access***
ROM AREA	0x8000-0x803F	0x00..0x3F	ROM	READ ONLY
A 256 Byte EEPROM (I^{2}C, C34CW02)	0x00..0x3F	0x00..0x3F	EEPROM	READ / WRITE
A 512 Byte EEPROM (SPI, M93C66)	0x00..0xDF	0x00..0xDF	EEPROM	READ / WRITE
A 8196 Byte EEPROM (I^{2}C, M24C64)	0x00..0xFDF	0x00..0xFDF	EEPROM	READ / WRITE

Table 10: EEPROM memory logical addressing scheme

[4] Must be zero!

2.2. The EEPROM API

EEPROMRead

Function:	Read one word (16-Bit) from a given EEPROM address.	
C-Prototype:	word EEPROMRead(word addr);	
Arguments:	addr	The logical address of the word to be read from.
Returns:		The value read.
Supported:	mCAT	All versions.
	Hardware	All
Comments:	*#include <nvmem.h> is required!*	

EEPROMWrite

Function:	Write one word (16-Bit) to a specified address of the EEPROM.	
C-Prototype:	void EEPROMWrite (word addr, word value);	
Arguments:	addr	The logical address of the word to be written.
	value	The value to be written
Returns:		
Supported:	mCAT	All versions.
	Hardware	All
Comments:	*#include <nvmem.h> is required!*	

EEPROMGetSize

Function:	Return the size of an EEPROM in units of 16-Bit words.	
C-Prototype:	*UINT16 EEPROMGetSize (UINT16 chip);*	
Arguments:	chip	The EEPROM Chip to be selected.
Returns:		Size of the read/write section in 16-Bit words
Supported:	mCAT	T00215 and higher
	Hardware	All
Comments:	*#include <nvmem.h> is required!*	

2.3. The mCAT V2 EEPROM usage

The first 16 EEPROM words are reserved for mCAT. Each value (except some reserved words) is listed below. Some parameters are spread over more than one EEPROM word as denoted in parentheses (), the name in [square brackets] is the name of the address definition in EEPROM.H.

Please note that EEPROM.H offers two macros to access LSB or MSB values of a word. Example:

```
node = GET_LSB(EEPROMRead( EE_BB_NODE_SPEED));
```

HARDWARE SERIAL NUMBER [EE_SER]

EEPROM words 0 + 1. It depends on the the manufacturer if a valid code is maintained or not. Do not overwrite this value.

BITBUS PARAMETERS (1) [EE_BB_NODE_SPEED]

NODE, LSB:

Node address. 0 is invalid, 0xff selects master mode.

SPEED, MSB:

0 = 62.5 kBit/s

1 = 375 kBit/s

2 = 750 kBit/s

3 = 1500 kBit/s

BITBUS PARAMETERS (2) [EE_BB_BUF_LEN]

MSGLEN, MSB:

The minimum value is 20, the maximum is 255.

BUFFERS, LSB:

The standard value is 8 for a slave and 32 for a master.

You may need more system memory for a master, see „EE_RAM“ for

Details.

SERx (1) [EE_Sx_SPEED] & SERx (2) [EE_Sx_BPC_PARITY]

See Serial driver documentation “*driver.pdf*” for details. The layout of these cells may be subject to of change in future versions.

FLASH CLEANUP VECTOR [EE_FLASH_CLEAN_A, EE_FLASH_CLEAN_B]

See chapter 2 for more details.

SYSTEM HEAPSIZE [EE_RAM]

This word holds the size of the system heap in kBytes. The standard value for a BITBUS master is „64 [dec]“ and „32 [dec]“ for a slave.

BIT900 FIFO DRIVER (1) [EE_FIFO_BUFFERS]

Number of buffers for the FIFO driver of the BIT900. These values may be used for other purposes on different hardware.

BIT900 FIFO DRIVER (2) [EE_FIFO_LEN]

Length of a buffer for the FIFO driver of the BIT900. These values may be used for other purposes on different hardware.

EE_LANGUAGE

This cell is used to code the used language. By default, this cell is not set.

EE_COUNTRY

This cell is used to code the country. By default, this cell is not set.

EE_CHARSET

This cell is used to code the used charset. By default, this cell is not set.

EE_ETHERNET_MODE

This cell is used to set the ETHERNET interface to a fixed mode. Possible settings are:

AUTONEGOTIATION	1	(or 0xFFFF, default)
10-BASE-T FULL DUPLEX	2	
10-BASE-T HALF DUPLEX	3	(default on NETA7)
100-BASE-TX FULL DUPLEX	4	
100-BASE-TX HALF DUPLEX	5	

EE_IP_DNS_0, EE_IP_DNS_1, EE_IP_0, EE_IP_1,
EE_IP_SUBMASK_0, EE_IP_SUBMASK_1, EE_IP_GATEWAY_0, EE_IP_GATEWAY_1, EE_IP_ETH_MTU, EE_SOCKETS

These values are needed for the Internet setup.

EE_BASIC_START

From EE_BASIC_START to EE_USER_START is the reserved area for the ELZET80 em-BASIC.

EE_USER_START

Define your own cells from EE_USER_START on:

```
#define MY_NEW_CELL       EE_USER_START

#define MY_NEXT_CELL      EE_USER_START+1
```

Please note that with a standard EEPROM only 16 cells are available to application use.

3. Accessing Flash Memory

3.1. DELPAGE

Erasing a page in a flash memory:

- is a time consuming process
- usually disables the access to the rest of the flash memory.

Therefore we have to take a few precautions when we remove something from the flash:

- Make sure the code to erase the flash is moved to and executed in RAM!
- Make sure that there is no task or interrupt driver running out of the page to be erased. System would crash!
- Make sure that the system integrity is not violated. Components vital for other components maybe located in the page to be erased. System may react unforeseeable!

To provide this, mCAT uses an approach called DELPAGE. The idea is to mark pages (up to 16 pages are supported) using an easy to alter bitmap in the configuration EEPROM. When a RESET or power up happens, the Non-Volatile Memory Management will delete all marked pages, clear the bitmap and restart the system to provide a clean system.

To provide a reliable operation the DELPAGE bitmap is stored twice! The second copy is inverted and only if a bit is set in one bitmap AND reset in the other, the page is really erased.

The DELPAGE command of SYSMON is used to alter the EEPROM bitmap. It expects up to 16 ordinal page numbers.

For the page number / address relationship, please refer to the following table:

DelPage #	***29F040***	***29F800T***	***2*29F800T****	***2*29F160****	***29F160***
	0.5MB	*1MB*	*2MB*	*4MB*	*2MB*
	TSMCPU900 TSMCPU8H2	NET900H NET900H+ PERDINX BIT900A BIT900PCI BIT900104	TSMCPU32H2	TSMCPUARM	NETA7
0	800000-80ffff	800000-80ffff	800000-81ffff	800000-83ffff	000000-01ffff
1	810000-81ffff	810000-81ffff	820000-83ffff	840000-87ffff	020000-03ffff
2	820000-82ffff	820000-82ffff	840000-85ffff	880000-8bffff	040000-05ffff
3	830000-83ffff	830000-83ffff	860000-87ffff	8c0000-8fffff	060000-07ffff
4	840000-84ffff*	840000-84ffff	880000-89ffff	900000-93ffff	080000-09ffff
5	850000-85ffff*	850000-85ffff	8a0000-8bffff	940000-97ffff	0a0000-0bffff
6	860000-86ffff*	860000-86ffff	8c0000-8dffff	980000-9bffff	0c0000-0dffff
7	-	870000-87ffff	8f0000-8effff	9c0000-9fffff	0f0000-0effff
8	-	880000-80ffff	900000-91ffff	a00000-a3ffff	100000-11ffff
9	-	890000-81ffff	920000-93ffff	a40000-a7ffff	120000-13ffff
10	-	8a0000-82ffff	940000-95ffff	a80000-abffff	140000-15ffff
11	-	8b0000-83ffff	960000-97ffff	ac0000-afffff	160000-17ffff
12	-	8c0000-84ffff*	980000-99ffff*	b00000-b3ffff*	180000-19ffff*
13	-	8d0000-85ffff*	9a0000-9bffff*	b40000-b7ffff*	1a0000-1bffff*
14	-	8e0000-86ffff*	9c0000-9dffff*	b80000-bbffff*	1c0000-1dffff*
15	-	8f0000-8fbfff*	9e0000-9effff**	bc0000-bfffff*	1e0000-1effff*

Table 11DELPAGE NUMBERS

* Pages reserved for mCAT

3.2. FLASH Memory API

FLASHWrite

Function: Copies *len* bytes from a memory location *src* to a flash location *dest*. The destination area must be empty (all 0xff).

The length must be a multiple of 4 and dest & src must be aligned to a 4 byte boundary!

C-Prototype: *INTEGER FLASHWrite (UINT32 *dest, UINT32 *src, UINT32 len);*

Arguments:

dest	Pointer to the flash
src	Pointer to the data
len	Length of data to be written (in bytes!)

Returns: TRUE on success

Supported:

mCAT	All, Need to align src/dest since T00215 for better inter platform compatibility.
Hardware	All

Comments: *#include <nvmem.h> is required*

FLASHErase

Function: Erase a page from flash memory

C-Prototype: *INTEGER FLASHErase (UINT32 *dest);*

Arguments:

dest	Pointer into the flash

Returns: TRUE on success

Supported:

mCAT	All
Hardware	All

Comments: *#include <nvmem.h> is required*

FLASHGetTypeCode

Function: Retrieve the FLASH identification code.

C-Prototype: *UINT32 FLASHGetTypeCode(UINT32 *flash);*

Arguments: *flash* Pointer to the flash

Returns: The upper 16-bit return the number of flash memories used in parallel to serve the systems bus size (example: 4 29F040 (single byte each) needed to feed a 32-Bit bus, 1 29F800 (two byte bus) needed to feed a 16-Bit system. Among others this information is needed to calculate *del-page* information.

The lower 16-Bit hold the manufacturer and the type code.

Supported:

mCAT	All
Hardware	All

Comments: *#include <nvmem.h> is required*

FLASHGetName

Function: Convert the id code information retrieved using *FLASHGetTypeCode* into a formatted ASCII string.

C-Prototype: *char *FLASHGetName(UINT32 typecode);*

Arguments: *typecode* Value returned by *FLASHGetTypeCode.*

Returns: Pointer to the ASCII representation of the flash type code or NULL if the chip is not known.

Supported:

mCAT	New since T00215!
Hardware	All

Comments: *#include <nvmem.h> is required.*

The system maintains a database of supported flash memories. This function is used to query this database!

FLASHGetNameByIndex

Function: Enumerate available flash memory devices.

C-Prototype: *char *FLASHGetNameByIndex(UNSIGNED index);*

Arguments: *index* *A positive integer, an index to the flash memory database.*

Returns: Pointer to the ASCII representation of the flash type code or NULL if index does not select a valid database entry.

Supported:

mCAT	New since T00215!
Hardware	All

Comments: *#include <nvmem.h> is required.*

The system maintains a database of supported flash memories. This function is used to query this database!

FLASHGetSize

Function: Retrieve the size of a flash memory.

C-Prototype: *UINT32 FLASHGetSize(UINT32 typecode);*

Arguments: *typecode* Value returned by *FLASHGetTypeCode.*

Returns: Size of a flash memory in kByte or 0 if this flash memory type is not known.

PLEASE NOTE THAT THE NUMBER OF CHIPS IS NOT TAKEN INTO ACCOUNT! THIS FUNCTION RETURNS THE RAW SIZE OF A FLASH MEMORY!

Supported:

mCAT	New since T00215!
Hardware	All

Comments: *#include <nvmem.h> is required.*

The system maintains a database of supported flash memories. This function is used to query this database!

FLASHGetSizeByIndex

Function: Enumerate available flash memory devices.

C-Prototype: *char *FLASHGetSizeByIndex(UNSIGNED index);*

Arguments: *index* *A positive integer, a index into the flash memory database.*

Returns: Size of a flash memory in kByte or 0 if index does not select a valid database entry.

Supported:

mCAT	New since T00215!
Hardware	All

Comments: *#include <nvmem.h> is required.*

The system maintains a database of supported flash memories. This function is used to query this database!

Example:

```
void flash_info()
{
       UINT32       id;
       UINT32       size;
       UNSIGNED     index;
       UNSIGNED     chips;


       id = FLASHGetTypeCode(0x800000); // assume flash at 0x800000
       chips = id >> 16;
       size = FLASHGetSize(id);
       if (id && size) {
              // a known flash
              kprintf(“FLASH AT 0x800000 is a %s, chip size is %lu ”
                     “system size is %lu [%u chips]\n”,
                     FLASHGetName(id),
                     size,
                     size*chips,
                     chips);
       } else {
              // well, lets look what we know ...
              index=0;
              do {
                     size = FLASHGetSizeByIndex(index);
                     if (size) {
                            // now we can be sure a description for this
                            // flash exists, so we can call FLASHGetNameByIndex()
                            // without further checking!
                            kprintf(“WE KNOW %s, its size is %lu\n”,
                                   FLASHGetNameByIndex(index),
                                   size);
                            index++;
                     } else {
                        index = 0;
                    }
            } while (index);
    }
```

3.3. Supported Flash Memories [08/01/2008]

Please note that not all devices „supported“ are „tested“. MOCOM tries to test every chip in the list and to add new types - however, it takes some time ...

Manufacturer	***Type code***	***Supported***	***Tested***
AMD	Am29F010	YES	YES
	Am29F040	YES	YES
	Am29F100	YES	YES
	Am29F200	YES	-
	Am29F400	YES	-
	Am29F800	YES	YES
	Am29F160	YES	YES
Fujitsu	MBM29F010	YES	-
	MBM29F040	YES	-
	MBM29F200	YES	-
	MBM29F400	YES	-
	MBM29F800	YES	-
ST	M29F040	YES	YES
	M29F100	YES	-
	M29F200	YES	-
	M29F400	YES	-

Manufacturer	***Type code***	***Supported***	***Tested***
HYUNDAI	HY29F040	YES	-
	HY29F200	YES	-
	HY29F400	YES	-
	HY29F800	YES	-
	HY29F160	YES	YES
MACRONICS	MX29F800T	MCAT2.1	YES
	MX29F004	MCAT2.1	-

4. I^{2}C-Driver

The I^{2}C[5]-Driver shall enable users to easily access any I^{2}C-Device in the system.

The I^{2}C-Driver has not always been reentrant and its use was restricted to single task use.

With T00215 the I^{2}C-Driver is reentrant on all platforms now.

4.1. I^{2}C Addressing

An I^{2}C-Device 7-Bit address is formed as follows:

$A_3..A_6$	$A_0..A_2$	R/W

A3..A6 : Device type address, usually defined by the chip manufacturer.

The Pattern 1111 for A3..A6 is reserved and should not be used in 7-Bit addressing mode.

A0..A2 : The Device address, usually selected using 3 Pins at the I^{2}C-Device.

R/W : 1 = READ ACCESS, 0 = WRITE ACCESS

4.1.0.1. Extended addressing

[5] I^{2}C is a registered trademark of Philips Semiconductors

1 1 1 1 0 A_9 A_8	R/W	8-Bit ADDRESS (A_0..A_7)

The 10-Bit addressing is not tested yet but implemented. Extended addressing uses the previously "reserved" pattern "1111xxxx" to transfer two address bits, BIT 9 (A_9) and BIT 8 (A_8) of the 10-Bit Address. The lower order bits are transferred in an extra address extension byte.

Extended (10-Bit) and normal (7-Bit) addressing are compatible and can be mixed in an I^{2}C-System.

4.2. I^{2}C-Function Reference

I2CReceive

Function: Receive *ilen* bytes from the I2C-Device addressed by *chip*.

C-Prototype: INTEGER I2CReceive (UINT16 chip, UINT8 *in, UINT16 ilen);

Arguments:

chip The I^{2}C chip address. If the MSB of chip is zero, standard 7-bit addressing is used. If the MSB is not zero, the three lowest bits of the MSB are used to extend the 7-Bit address in the LSB. Please note that the 7-bit address must be given in I^{2}C format including the read/write bit (bit 0). However, you must not worry about the R/W bit – the driver sets it up as needed.

in Pointer to the receive data buffer in RAM.

ilen The size of the receive buffer in bytes.

Returns: TRUE if read was successful

Supported:

mCAT	All
Hardware	All

Comments: *#include <nvmem.h> is required.*

I2CReceiveEx

Function: Send *olen* bytes and then receive *ilen* bytes from the I2C-Device addressed by *chip*. This is a combination of the functions *I2CSendEx* and *I2CReceive*. This function executes both functions with tasking blocked to ensure integral execution.

C-Prototype: INTEGER I2CReceiveEx (UINT16 chip, UINT8 *out, UINT16 olen, UINT8 *in, UINT16 ilen);

Arguments:

chip The I^{2}C chip address. If the MSB of chip is zero, standard 7-bit addressing is used. If the MSB is not zero, the three lowest bits of the MSB are used to extend the 7-Bit address in the LSB. Please note that the 7-bit address must be given in I^{2}C format including the read/write bit (bit 0). However, you must not worry about the R/W bit – the driver sets it up as needed.

out Pointer to the send buffer

olen Number of bytes to be send from the send buffer

in Pointer to the receive data buffer in RAM.

ilen The size of the receive buffer in bytes.

Returns: TRUE if read was successful

Supported:

mCAT	All, new in T00215
Hardware	All

Comments: *#include <nvmem.h> is required.*

I2CSend

Function: Send o*len* bytes to the I2C-Device addressed by *chip*.

C-Prototype: INTEGER I2CSend(UINT16 chip, UINT8 *out, UINT16 olen);

Arguments:

chip	The I^{2}C chip address. If the MSB of chip is zero, standard 7-bit addressing is used. If the MSB is not zero, the three lowest bits of the MSB are used to extend the 7-Bit address in the LSB. Please note that the 7-bit address must be given in I^{2}C format including the read/write bit (bit 0). However, you must not worry about the R/W bit – the driver sets it up as needed.
out	Pointer to the send buffer
olen	Number of bytes to be send from the send buffer

Returns: TRUE if send was successful

Supported:

mCAT	All
Hardware	All

Comments: *#include <nvmem.h> is required.*

I2CSendEx

Function: Send o*len* bytes to the I2C-Device addressed by chip but don't send terminating I2C-STOP condition. This function is needed when data must be send first to get a proper response. A I2CReceive call *must* follow a call to I2CSendEx!

C-Prototype: INTEGER I2CSendEx(UINT16 chip, UINT8 *out, UINT16 olen);

Arguments:

chip	The I^{2}C chip address. If the MSB of chip is zero, standard 7-bit addressing is used. If the MSB is not zero, the three lowest bits of the MSB are used to extend the 7-Bit address in the LSB. Please note that the 7-bit address must be given in I^{2}C format including the read/write bit (bit 0). However, you must not worry about the R/W bit – the driver sets it up as needed.
out	Pointer to the send buffer
olen	Number of bytes to be send from the send buffer

Returns: TRUE if send was successful

Supported:

mCAT	All
Hardware	All

Comments: *#include <nvmem.h> is required.*

XIII. mCAT Tools Documentation

1. The Basic Structure of mDE

There are several tools supplied with mDE. For the daily work at mCAT projects, there are only two tools of importance: the special ***make*** utility **vmake.exe** and the ***Terminal program*** **wLGO**.

At least **vmake** relies on a set of small helper routines and this documentation is about the relation and the communication between these tools.

1.1. The Folder Structure

The mCAT folder tree looks like this:

<mcatpath>\BIN32	Executables for Windows 95/98/NT
<mcatpath>\ETC	All configuration files (MCATINF.INI et.al.)
<mcatpath>\ETC\<core>	One subdir for each CORE is provided
<mcatpath>\DOC	Online doc as PDF files
<mcatpath>\CC	C-source tree
<mcatpath>\CC\INCLUDE	C-include files
<mcatpath>\CC\LIB\v4	C-librarys for TLCS900 C-Compiler 4.11
<mcatpath>\CC\LIB\V103	C-librarys for TLCS900 C-Compiler 1.03
<mcatpath>\CC\LIB\gnude	C-librarys for GNU C-Compiler (GNUDE, ARM)
<mcatpath>\CC\LIB\hcarm	C-librarys for Metaware C-Compiler (ARM)
<mcatpath>\CC\<examples>	C-Example code

1.2. MCATINF.INI

The mCATINF.ini file contains many variables used by mDE to locate files, locate directories and controlling mode. This file is an mCAT specific environment. All mocom tools refer to this file and use the information from mcatinf.ini to locate files and to select specific modes.

A WINDOWS® registry entry is used to refer to mcatinf.ini. So it is possible to have serveral different mcat installations and setups on one computer and to switch between the different installations just by changing this registry entry.

The registry location used has been changed with mCAT 2.20 to better support users who do not have the access rights to add or change entries in HKEY_LOCAL_MACHINE.

1.2.1. Registry Entry before mCAT2.20

```
HKEY_LOCAL_MACHINE/Software/mocom/mCAT/Path
```

1.2.2. Registry Entry with mCAT2.20

```
HKEY_CURRENT_USER/Software/mocom/mCAT/Path
```

1.2.3. Section [MCAT] in mCATINF.INI

ALL VARIABLES IN THIS SECTION CAN BE USED WITHIN MAKEFILES BY JUST USING THE VARIABLE NAME AS A MACRO NAME!

One example is the macro $(HARD), which is read from the MCATINF.INI file. However, all variables can be overwritten by macro definitions within makefile (The re-definition of macros inside a makefiles will have no effect on mcatinf.ini).

Variable	***Type***	***Description***
BIN	PATH*	The path to the mcat binaries.
CINCLUDE	PATH	The path of the mCAT C-include folder.
CLIB	PATH	The path of the mCAT C-runtime library folder.
AINCLUDE	PATH	The path of the mCAT ASM-include folder.
ALIB	PATH	The path of the mCAT ASM-runtime library folder (currently not used).
THOME	PATH	The path to the C-Compiler home
MAKEINI	FILE**	The name of the vMake default macro file (v4.ini for C V4.1 / v103.ini for C V1.03)
TINCLUDE	PATH	The path of the Compilers C-include folder.
TLIB	PATH	The path of the Compilers C-runtime library folder.
HARD	CONST***	Defines the „core“ used (CPU type, memory layout etc.)
CORE	CONST	Defines the „hardware“ used (external IO)
PFAM	CONST	Defines the processor family (TLCS900 or ARM7) *DO NOT CHANGE!!*
VERSION	CONST	Defines the current major mcat version (“2.20”). *DO NOT CHANGE!!*
std_ram	QFILE****	The standard ram target (defines the base address for programs) used for the examples. Can be redirected. See „5.2.5. MKLNK“ for more information on TARGETS.
std_rom	QFILE	The standard rom target (defines the base address for programs) used for the examples. Can be redirected. See „5.2.5. MKLNK“ for more information on TARGETS.
linktmpl	QFILE	The currently used linker template file. See „3.2.2. MKLNK“ for more information on linker templates

Table 12: mCATINF.INI Environment Variables

[*] A path only (without a file name)

[**] A file name only (without a path)

[***] A constant definition (can be passed to C-Compiler as a macro)

[****] A qualified file (path + file name)

The file can be manipulated by use of a std. ASCII-editor, the SETVAR command (3.5.1.) or by your own programs using the Windows functions „WritePrivateProfileString“ and „GetPrivateProfileString“.

1.2.4. Path macro replacement

It is allowed and recommended to use path macros within mcatinf.ini. Currently there are only a few macros defined:

Macro	***Description***
$(HOME)	Expanded to the fully qualified mCAT home path
$(THOME)	Expanded to the fully qualified C-Compiler home path
$(BIN)	Expanded to the fully qualified mCAT binary path
$(ETC)	Expanded to the fully qualified mCAT ETC path (where mactinf.ini itself is stored)
$(CORE)	Expanded to a fully qualified path to the core file folder. Within the core file folder, the core specific target files for the given core are stored (see „std_rom“/“std_ram“)

Table 13: Path Replacement Variables

Using the macro syntax, you can also refer to other macros declared inside mcatinf.ini:

```
THOME=c:\gnude
TINCLUDE=$(THOME)\include
```

Please note that these macros use the same syntax the makefile macros use. However, the macros are resolved on a lower level than the macros „vMake“ uses. Make will always read the expanded value when reading a variable from mCATINF.INI.

With mCAT 2.20 it is also possible to refer to environment variables. Example:

The GNUDE GNU-Compiler set exports the environment variable

```
GNUDE=<gnude-path>
```

In the mCATINF.INI we can use

```
THOME=$(GNUDE)
```

to refer to the GNUDE path, in all subsequent macros in mcatinf.ini **AND** in the makefiles and vmake ini-files where THOME is referred to. THOME is then replaced by the value of the environment variable $(GNUDE).

1.3. MCATPATH

MDE uses one fundamental variable to access all information needed. This variable is the mCAT HOME PATH! In one or the other way, mDE tools must know this home path to work properly.

The mCAT home path is stored in „HKEY_LOCAL_MACHINE\\Software\\mocom\\mCAT“ as item „Path“.

2. The Tools

2.1. CIMD

The CIMD.EXE program is a generator for IMD-Files. From a set of command line arguments, it generates a “C” source code module that includes an IMD structure and all values needed to initialize the IMD. The advantage of cimd.exe over the older imd.exe is that it can be easily integrated into a projects makefile.

Note: Existing Projects will compile without changing the IMD. Use CIMD for new projects only!

```
 *** CIMD 1.00 ***
cimd {options} <name> <imdfile.c>

options are:
 -public         : imd structure is public (default private)
 -auto           : autostart option, default is 'no autostart'
 -interrupt      : setup imd mode MODE_INTERRUPT (default=MODE_TASK)
 -initmodule     : setup imd mode MODE_INIT (default=MODE_TASK)
 -version=#.##   : set initial version, default=1.00
 -priority=###   : set initial priority to ### (0 < priority <255), default=128
 -stack=####     : set stack size, default is 1024
 -init=<name>    : name of the init call, default is __cstart
 -main=<name>    : name of the main call, default is TaskMain
 -imd=<name>     : imd structure name, default is taskimd
```

This fragment from a *makefile* shows the usage:

```
OBJFILES = imd.$(REL) $(PROJECT).$(REL)

imd.$(REL):         imd.c               $(INCFILES)
imd.c:       makefile
      $(CIMD) -public -auto -init=NULL -version=1.00 mCAT/XPSERVER imd.c
```

Every time the makefile is changed (maybe because the *version* argument of cimd.exe was changed) the xpimd.c is generated from the command line arguments.

Make sure your generated imd file is the first in the list of modules to link!

2.2. IMD

The IMD.EXE program is obsolete, use CIMD for new projects.

2.3. VMAKE

VMAKE is an UNIX „make“ type program to control the compilation and linking process. If not specified differently, vMake will only compile those modules that have been changed since the last call of vMake. With bigger projects, that will reduce turn-around times dramatically.

The duties of vMake are:

1. Executing the C-compiler and / or assembler as needed.
2. Pass arguments to compiler / assembler
3. Generate a linker control file using MKLNK.EXE
4. Link the project
5. Convert the resulting binary file to a hex file (S3/S7, „SHX“)
6. Tag the generated SHX file using S3PATCH.exe

2.3.1. Using Macros

Macros are essential for the understanding and use of vmake.

Defining a macro:

```
<macro-name> = value
```

Example:

```
PROJECT = thread
```

Referring a macro:

```
$(<macroname>)
```

Example:

```
OBJFILES = $(PROJECT).$(REL)
```

After replacement, the line will be: *OBJFILES = thread.$(REL)*

This documentation is not a full introduction to the general possibilities of a make program. However, pre-defined macros, mCAT specific extensions and some other extra commands not common with „classic“ 'make' will be described.

2.3.2. Command Line Format

```
vMake {-f <makefile>} {flags} {macro=value}
```

Macro definitions can be any valid make macro. The command line macros can be used like any other makefile macros using the $(<macro-name>) syntax.

Argument	Comment
-a	Re-make target. With a typical mCAT makefile, a re-link of the target is made - but no compilation if there is no need.
-r	Re-make all unconditionally
-f	<makefile>. If no makefile is given, the default makefile „makefile“ is used.
-d	Display the reasons why MAKE chooses to rebuild a target. All dependencies which are newer are displayed
-dd	Display the dependency checks in more detail. Dependencies which are older are displayed, as well as newer.
-D	Display the text of the makefiles as read in.
-DD	Display the text of the makefiles and 'make.ini' (v4.ini / V103.ini).
-e	Let environment variables override macro definitions from makefiles. Normally, makefile macros override environment variables. Command line macro definitions always override both environment variables and makefile macro definitions.
-i	Ignore error codes returned by commands. This is equivalent to the special target .IGNORE:.
-k	When a non-zero error status is returned by a command, abandon work on the current target, but continue with other branches that do not depend on this target.
-n	No execution mode. Print commands, but do not execute them. Even lines beginning with an @ are printed. However, if a command line is an invocation of MAKE, that line is always executed.
-q	Question mode. MAKE returns a zero or non-zero status code, depending on whether or not the target file is up to date.
-s	Silent mode. Do not print command lines before executing them. This is equivalent to the special target .SILENT:.
-S	Undo the effect of the -k option. Stop processing when a non-zero exit status is returned by a command.
-t	Touch the target files, bringing them up to date, rather than performing the rules to reconstruct them.

Table 14: vmake Command line arguments

2.3.3. DEFAULT Makefile: v4.ini / v103.ini

VMAKE will read a „default“ file prior to reading - and executing - the makefile. Depending on the installed c-compiler, this file is V4.ini (CC 4.0 and higher) or V103.ini (CC 1.03). The file supplies some macro and rule definitions. For example, the default rules to compile c- and asm-files are included.

To be able to customize the compilation process, a few macros are included in the rule definition. These can - but don't need to - be defined by the user.

Macro	*Example*	*Comment*
CMD	CMD=-D$(CORE) -D$(HARD)	Add flags and definitions to the c-compilers command line
APPCMD	APPCMD=-D$(CORE)	Add flags and definitions to the assembler preprocessors command line

Table 15: Passing extra options

2.3.4. The Basic Structure of an mCAT Makefile

We use the makefile from the thread example to describe the basic structure.

The header includes some comments. We use PRINT instead of real comments, so that the comment will be displayed every time the makefile is invoked.

```
PRINT
PRINT  ** THREAD PROGRAMMING EXAMPLE **
PRINT
PRINT  (c) 1999 mocom software GmbH & Co KG
PRINT  Author:      Volker Goller
PRINT
```

The PROJECT macro defines the project name used by several tools. This is the name of the main „c“ („$(PROJECT).c“) or „asm“ („$(PROJECT).s“) source file as well as the name of the hex file („$(PROJECT).shx“) file to be generated.

```
PROJECT = thread
```

The TARGET macro is used to select an address descriptor file (target file, „.trg“). If the target file $(TARGET).trg exists in the same sub-dir the makefile is located, it is used. If not, it is checked if TARGET is an item within the mCAT section of the „mcatinf.ini“ file. In that case, the value of this item is the file name of the target file to be used.

In the *thread* examples makefile the value **std_ram** is assigned to the macro TARGET. std_ram can be found in the *mcatinf.ini* file.

```
TARGET = std_ram
```

You may want to pass the ***core*** name as a preprocessor macro to the c-compiler. That allows you to use the C preprocessor to customize you program code depending on the core hardware it should work with. To do that, we use the CMD macro:

```
CMD = -D$(CORE)
```

Within your C code, you can simply write thinks like:

#ifdef **NET_H** /* CORE = NET_H: CPU is TMP95C063 */

#include <t95c063.h>

#endif

The OBJFILES macro is a list of all user supplied object files that are needed to correctly link the project. If more than one object is used, just add them to this line (separated by a space). **If CIMD is used to create an imd file, the imd object must be the first in the list. See CIMD.**

> *Changed: In mCAT Environment 2.20 the ".rel" extension in makefiles is replaced by the macro ".$(REL)". This allows it to easily adapt the environment to different C-Compiler packages.*

```
OBJFILES = imd.$(REL) $(PROJECT).$(REL)
```

The list of include files is used to describe dependencies other than the relation of source to object files. One example would be a commonly used header file or the makefile itself! Whenever you change makefile, a recompile would be forced!

```
INCFILES =
```

The main rule describes how to make a hex file out of the object files if one or more object files are newer than the hex file. It takes four steps to make the hex file out of the objects:

1. Generate a linker control file using MKLNK.EXE
2. Link the project using the Toshiba linker
3. Tag the generated ABS file using S3PATCH.EXE "$(S3PATCH)"

4. Convert the ABS file to a downloadable hex file (S3/S7, „SHX“)

```
$(PROJECT).shx: $(OBJFILES)
       $(MKLNK) $(TARGET) $(PROJECT) $(OBJFILES)
       $(LD) $(PROJECT).LNK -e LibInit -o$(PROJECT).abs
       $(CONVERT) $(PROJECT).abs $(OF) $(PROJECT).shx
       $(S3PATCH) -l=$(PROJECT).map $(PROJECT).shx
```

Finally, we have to define the dependencies of the object files. The default entry is the entry for the main source file ($(PROJECT).c = thread.c). We include the *INCFILES* macro at the end of line. This will make sure that changes to the files included in the macro *INCFILES* will force a re-build, too.

```
$(PROJECT).$(REL):  $(PROJECT).c  $(INCFILES)
```

2.3.5. Extended Commands

Commands are build-in functions that extends the possibilities of targets & macros.

PRINT

PRINT

vMake 2.01 supports a PRINT command. It allows an user to print a text or macro value at runtime. The text is send to stdout. Example:

```
PRINT
PRINT ** Hello, this is a makefile **
PRINT
PRINT
PRINT CORE=$(CORE)
```

should output (assuming $(CORE) = FY64):

```

    ** Hello, this is a makefile **

    CORE=FY64
```

The PRINT command is a big help for debugging makefiles!

INCLUDE

The include command can be used to load additional macros from another file. The filename can be a constant text, a macro value or a combination of both:

```
include debug.mak
```

or

```
include $(HARD).mak
```

SETINF

vMake can read any item from the mCAT section of the **mcatinf.ini** file. All items read can be overwritten inside vMake by makefile macro definitions. But these changed values will not be written back to mcatinf.ini!

However, item values in **mcatinf.ini** can be changed from within a makefile by use of command SETINF.

The syntax is:

```
SETINF  <item>=<value>
```

If a value has to include a path macro instead of a specific path, the path macro must be prefixed by the „\“ backslash escape character!

```
SETINF set_ram=$(CORE)\std_ram.trg
```

will result in

```
set_ram=FY64\std_ram.trg
```

while

```
SETINF set_ram=\$(CORE)\std_ram.trg
```

will result in

```
set_ram=$(CORE)\std_ram.trg
```

2.3.6. MKLNK

MKLNK is a program that generates a linker control file for Toshibas Linker from a given template file and a target address descriptor file. The program was designed to fix some problems that arise while using Tohiba tools with vMake:

- With the Toshiba tools the command line is limited in size
- The addresses where to place data and code cannot be passed via command line
- The linker control file is pretty complex and feature-rich. However, for 99% of all mCAT applications there is no need to use more than just a few standard statements in the linker file.

MKLNK will create an ***<project-name>.lnk*** file from the template and the target file. MKLNK accepts a few arguments:

```
mklnk <project-name> <target-name> <obj-file> {obj_file} .. {obj_file}
```

Where *project-name* is the name of the main object file (see example *THREAD*). The *target-name* is the name (without extension) of the target file that should be used. If the target file (*<target-name>****.trg***) is available in the current directory it is used. If not, MKLNK will search the section „mCAT“ within ***mcatinf.ini*** for an item called *<target-name>.* The value of this item is a path to the target file (including path, filename and extension) that should be used. For example, the predefined targets ***std_rom*** and ***std_ram*** are defined there.

Within the linker template file (*default.lnk* for tools V4 and *default.v13* for tools V1.03) a few macros can be used.

Macro	***Comment***
@PROJECT	The name of the project from the command line
@OBJFILES	Will be replaced by the object files given on command line
@TLIB	The path to the C-Compiler runtime libraries read from mcatinf.ini
@CLIB	The path to the mCAT runtime libraries read from mcatinf.ini

Table 16: Macros used in a linker control file

Additionally, all items from the ADDR section of a target file can be used as a macro. Usually the following items are defined (and used by *default.lnk / default.v13*):

Macro	***INI Item***	***Comment***
@RAMSTART	ramstart	The first address in RAM the application can use
@RAMLENGTH	ramlength	The length of the RAM area the application can use
@ROMSTART	romstart	The first address in ROM the application can use
@ROMLENGTH	romlength	The length of the ROM area the application can use

Table 17: Address macros used in a linker control file

2.3.7. S3PATCH

The S3PATCH program reads the Motorola SS/S7 files generated by the compiler/linker and modify them in four ways:

1. Convert to a unique S3/S7 record length that is optimal for use with WLGO to improve download times. You can also choose other values for record length using the option -s=<reclen>. Example: if you download your programs using BAPIMON, use a record length of 240 for best performance.

2. Fills gaps in the S3/S7 records. The Toshiba compilers do not emit code if not needed. That makes the S3/S7 image very ugly to handle when you have to program FLASH memory 32-Bit wise. S3PATCH fills those gaps.
3. Insert current daytime into the IMD-Structures found in the image. This is an additional aid to provide compile time information at runtime. When using the *modules* command of SYSMON this compile time is displayed.
4. Insert ROM and RAM area information into the extended IMD sections. The information is usually extracted from the mapfile the linker generates.

```
 *** S3PATCH 1.02 ***

ERROR: Unknown argument

s3patch [options] <filename>

 's3patch' will patch and replace the given S3/S7 file.
 It replaces former tool 'tag.exe'

  -s=n:  n=S3 record length
  -b  :  Use a given time as build instead of the current machine time
  -k  :  Optional key.
  -r  :  Resize S3 record size. No other modification made.
  -d  :  RAM area. <base> is the base addr, <length> is the length.
  -c  :  ROM area. <base> is the base addr, <length> is the length.
  -t=<target_file>
      :  A optional mCAT target file containing RAM /ROM info

  -l=<link_file>
      :  A optional linker file containing RAM /ROM info

  -a={-}<offset>
      :  Add offset to emited addresses
```

2.3.8. TAG

TAG.EXE is obsolete, use S3PATCH instead.

2.4. Terminal program

The terminal program available is wLGO (32-Bit Windows95/98/NT). LGO is an abbreviation for „Load and Go“.

WLGO is a very basic implementation of a Terminal, just perfectly suited for mCAT development.

2.4.1. Features

- After startup the programs will check the current assigned serial line to identify the attached mCAT system.
- A SHX file can be loaded from command line or by use of **F3** key.
- Received data log file. This feature is toggled using the **F2** key
- A little help can be displayed using the **F1** key.
- **F4** will clear the current screen.
- A light weight ANSI Terminal emulation is integrated into both programs, too. The emulation is optimized and tested to work with the screen control macros defined in ANSI.H.
- The programs are using the mCAT baudrate (19200) by deafult.
- It is possible to PASTE text from the clipboard into a session. However, currently it is not possible to copy text back to the clipboard.
- The serial line options, the font and the default directory for the log files can be set via options from the main menu.

If you have to connect to another comport, you can change the port to use in the WLGO Menu “Einstellungen->Schnittstelle”.

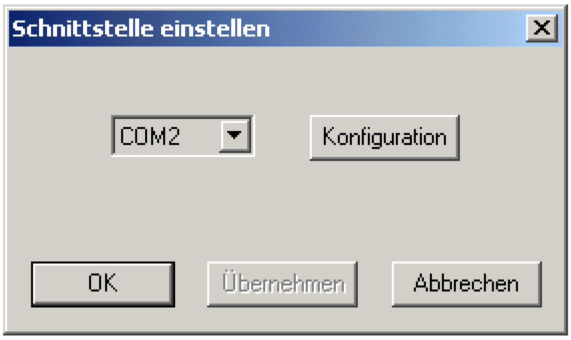
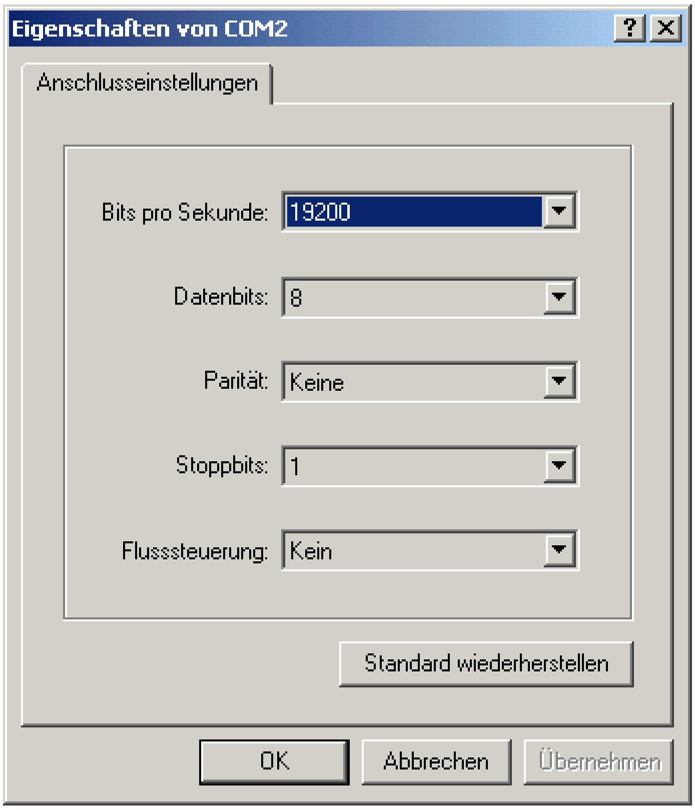


Figure 25: wLGO, Changing the COMPORT

We need 19200-8N1, no handshake, to communicate with mCAT. You will find the setup at “Einstellungen->Schnittstelle->Konfiguration” (the Button shown in the figure above).



Eigenschaften von COM2

Anschlusseinstellungen

Bits pro Sekunde: 19200

Datenbits: 8

Parität: Keine

Stoppbits: 1

Flusssteuerung: Kein

Figure 26: wLGO, Setting the COM parameter

2.5. Hex File Tools

2.5.1. HEX2IMG

HEX2IMG will convert an INTEL hex file or an MOTOROLA S3/S7 hex file into a binary image file. Several options are supported:

```
hex2img <hex-file> {flags}
```

Flag	Description
-E=<end>	Specifies the highest address allowed for the image. From last byte defined in the *hex file* until <*end*> the image is filled with 0xff.
-O=<offset>	Defines the start address or offset of the image. Imagine an SHX file defining the first byte at 0x800000. Without the option „-O=0x800000“, the output file would be at least 8MB (0-0x800000) in size!
-A	Auto offset. The image starts at the lowest address defined within the hex file.
-C	Same as –o=0x8000 –e=0xffff
-M	Generate map file (recommended). This option is very useful when used with „merged“ hexfiles (see hexmerge.exe).

Table 18: HEX2IMG Arguments

2.5.2. IMG2HEX

IMG2HEX does the opposite of HEX2IMG: It creates hex files form images.

```
img2hex <bin-file> <base_addr> {flags}
```

Flag	***Description***
<bin-file>	An binary image file - maybe generated using hex2img ...
<base-addr>	The base address or offset of the file. That is the physical address in memory of the first byte in the file.
–i	By default, img2hex generates Motorola S3/S7 files. Issuing a „–i“ flag will force INTEL extended hex format instead (the INTEL format is limited to addresses < 0x100000!).
–y	Generate an INTEL extended hex file and add the checksum over all data bytes to the end of file. LGO will be able to read this checksum. This option is used to generate files that can be downloaded via the TMP95FY64 rom-loader.

Table 19: IMG2HEX Arguments

2.5.3. HEXMERGE

HEXMERGE is used to create merged hex files. Merged hex files consist of several independent hex files into a single file, usually separated by lines starting with an at-character (@) and containing the path of the hex source file included next.

```
HEXMERGE {$} <destination-file> <{file-list} | {@file}>
```

All files given at command line behind destination are copied to destination. Instead of a list of files a file containing such a list (one file per line) can be used. In this case an at-character (@) must begin the list file name.

If a '$' is inserted as first argument the filenames of the modules are not emitted to the destination file.

Please note that if HEX2IMG is used with a merged hex file and generation of a MAP file is allowed, the mapfile will include all modules with path and size!

Examples:

hexmerge load app int shlib

will result in a single hex file called LOAD.SHX:

```
@app.shx
S3.....
@int.shx
S3...
@shlib.shx
S3...
S7...
```

If a text file (lets say merge.lst) exists containing:

```
app.shx
int.shx
shlib.shx
```

then an a alternate use of hexmerge is possible:

hexmerge load @merge.lst

2.6. Little Helpers

2.6.1. SETVAR

SetVar is used to change items in mcatinf.ini from a command line or from within a batch file:

```
SETVAR section item value
```

section: The section of mcatinf.ini, usually „mCAT“

item: Any item inside the given section. If the item does not exists, it will be created.

value: Any value you like to set. If value is a path (maybe including mCATINF path macros) or contains spaces, use quotation marks around the value.

Example:

```
      setvar mcat linktmpl "$(ETC)\default.lnk"
```

Sets the default linker template to „$(ETC)\default.lnk“ (for use with CC V4.xx).

2.6.2. MCATPATH

The mCATPATH registry entry can be displayed and changed using the mCATPATH.EXE tool:

```
MCATPATH {path}
```

If ***{path}*** is not given, the current path is displayed.

2.7. MIC

MIC is a compiler that generates a set of files needed to create SharedLibraries for mCAT. The files generated should NEVER be edited manually.

MIC does not support the Toshiba C-Compiler Version 1.03

2.7.1. The MIC Source File

Within a MIC source file contains information needed to generate:

- a function wrapper file (wrapper.c) to convert the mCAT calling convention into c calling convention (C only, with assembler we assume that your functions will use the mCAT calling conventions).
- an include file declaring the prototypes of the wrapper functions (wrapper.h).
- an init file containing the TaskInit function, a jump table, an IMD and all structures needed to attach and use a SharedLibrary.
- a common include file defining library internal structures and constants (common.h)
- The header files needed by an application to use the library functions (interface files)

The source file uses the extension „.ldf“ (library definition file).

Please see the shared library sample in „*cc\shlib*“ for details.

Comments

A comment is delimited by the '#' character and the end-of-line.

Example:

```
# THIS IS A COMMENT
```

FMT

The FMT (FORMAT) keyword allows you to control the “face” of the generated code and select some styles. The FMT statement should be the first statements in the file!

Modifiers:

1. NOLOCAL: No LOCAL structure is generated and it is not included as an argument. This is helpful if you want to keep your library code compileable under other operating systems.
2. NOSCORE: Generates underscores for the WRAPPER functions and none for the implementation functions. That is the opposite of the default!
3. USERLIB: This is a MUST for users! The user libraries use different ordinal numbers than the system libraries. System libraries are technically the same thing as user librarys – the only difference is that system libraries are reserved for system extension by MOCOM.

LIBID

The LIBID statement defines a name for the library. This name is used to register the library in the USRLIB.INI file ($(ETC)\usrlib.ini. The major library number is allocaed by use of this name.

Example:

```
LIBID MyShLib
```

VERSION

The version statement allows to control the version information of the Library's IMD. The Version should be incremented with each change to the LDF file or the shared library project.

Example:

```
VERSION 1.00
```

ARG

A few command line arguments can be set using the ARG statement.

Keyword	***Argument***	
NAME	-n=<name>	give an IMD name (max. 16 char)
USEINIT	-u	call user init code "InitLibrary(LOCAL *local)"
OLDSTYLE	-l	LIBDEF compatible mode: • no wrappers (wrapper.c, wrapper.h) • no <libname>.s stub but <libname>.inc
AINCLUDE	-ia=<path>	set alternate ASM include path (for test etc.)
CINCLUDE	-ic=<path>	set alternate C include path (for test etc.)

Table 20: MIC, the ARG argument

Example:

```
ARG NAME=Sample/ShLib AINCLUDE=include CINCLUDE=include USEINIT
```

COMMON / ENDCOMMON

The text between COMMON and ENDCOMMON is copied to common.h without modification. Place internal structure and constant definitions HERE!

Example:

```
COMMON
    typedef struct {
        long    cnt;
    } LOCAL;
ENDCOMMON
```

Please note that the INCLUDE / ENDINCLUDE section (see next chapter) is copied to common.h as well!

INCLUDE / ENDINCLUDE

The text between INCLUDE and ENDINCLUDE is copied to *<ldf-file-name>.h.* Place all public structure and constant definitions here. This is the interface to the library!

If the modifier **@ASM** is used behind INCLUDE, the text between INCLUDE @ASM and ENDINCLUDE is copied to *<ldf-file-name>.def.*

MIC uses the default include directories from mcatinf.ini to place the interface files right there! If you want mic to use different directories, use the command line options –ic/-ia or the ARG statement!

Example:

```
INCLUDE
    typedef struct {
        short   len;
        char    string[1];
    } LSTRING;
ENDINCLUDE
```

LIBDEF

The mCAT SharedLibrary naming conventions recommend that every function should start with a short „Family“ pattern - Modula-2 programmers know this pretty well! The LIBDEF statement will define the family pattern for the next functions declared behind this statement - until next LIBDEF.

The syntax used to declare function prototypes is pretty similar to the C syntax, with a few exceptions:

1. The function name comes first
2. The prototype is introduced with a '?'
3. If the functions will be implemented in assembler, a **@ASM** modifier may follow. This will exclude this function from the wrapper files!
4. The pre-defined functions RESERVED just reserves a function entry for future use. This will keep libraries binary compatible.
5. If no “LIBDEF” prefix should be used, use the modifier anonymous!

Example:

```
LIBDEF Swap # assembler functions
    Word64  ? @ASM void  (void *w64);
    Word32  ? @ASM void  (void *w32);
    Word16  ? @ASM void  (void *w16);
    RESERVED
    RESERVED
LIBDEF Statistics                # implements a library invokation statistic
    GetCounter ? short (void);   # Use: short StatisticsGetCounter (void);
LIBDEF ANONYMOUS
    keypressed ? int (void);     # Use: int keypressed (void);
```

END

A LDF file must be terminated with an explicit END statement.

Example:

```
# NEVER FORGET ...
END
```

2.7.2. The MIC Command Line Arguments

Argument	
-g	generate all library files (wrapper.c, common.h, ...)
-i	generate the interface files (<ldf-name>.h, <ldf-name>.def)
-y	override all. If not specified MIC will query before overwriting existing files.
-c	library will be written in C
-asm	library will be written in assembler
-n=<name>	give an IMD name (max. 16 char)
-u	call user init code "InitLibrary(LOCAL *local)"
-l	LIBDEF compatible mode: • no wrappers (wrapper.c, wrapper.h) • no <libname>.s stub but <libname>.inc
-ia=<path>	set alternate ASM include path (for test etc.)
-ic=<path>	set alternate C include path (for test etc.)
-h	Used for all non-Toshiba targets. This option is usually automatically set when using the $(MIC) macro to execute ***mic*** from a makefile.

Table 21: MIC, command line arguments

3. Creating own Projects

An own program should be located in its own sub-dir, too. Or within any other private directory.

3.1. Creating a Makefile and Executing the Program

- Copy **„makefile“** template from „<mcatpath>\cc“.
- Load makefile into a program editor.
- Set project name to „myfirst“
- Set target to „std_ram“
- Save makefile and exit the editor
- Execute **vMake**
- Download myfirst.shx using wLgo / Lgo
- Use „init 402000“ to start the module

3.2. Creating a TARGET File

If several independent modules should be used, std_ram and std_rom targets can be used as a template for own target files. If the modules should be useable on different hardware, create new targets within the core sub-directories (<mcatpath>\etc\<core_name>) as the std_rom/std_ram does.

XIV. mCAT Release Documentation

1. MCAT 2.20

1.1. Porting Existing Applications

This is the first portable mCAT Version. The current release is available on ARM7 Microprocessor architecture platform – the TSMARMCPU. There are a few things to consider and to take care of when porting an existing mCAT application to mCAT 2.20.

1.1.1. Changes Needed

There are only a few changes needed to port existing mCAT applications to 2.20/ARM7 – As long as they do not use assembly code or hardware related stuff. If your application uses ExpressIO for process i/o access, it should be easily portable.

1.1.1.1. The Makefile(s)

Makefiles changed a bit to support the GNU and Metaware C-Compilers. First, replace the ".rel" extensions in your makefile by ".$(REL)". This macro will be automatically replaced by a suitable extension at runtime. Second, exchange the linking sequence. Your existing may look like this:

```
$(PROJECT).shx: $(OBJFILES)
      $(MKLNK) $(TARGET) $(PROJECT) $(OBJFILES)
      $(LD) $(PROJECT).LNK -o$(PROJECT).abs
      $(TAG)
      $(CONVERT) -o $(PROJECT).shx $(PROJECT).abs
```

To be conform with the mCAT2.20, replace this sequence with:

```
$(PROJECT).shx: $(OBJFILES)
      $(MKLNK) $(TARGET) $(PROJECT) $(OBJFILES)
      $(LD) $(PROJECT).LNK -o$(PROJECT).abs
      $(CONVERT) $(PROJECT).abs $(OF) $(PROJECT).shx
      $(S3PATCH) -l=$(PROJECT).map $(PROJECT).shx
```

1.1.1.2. The C-Source

The C-Source needs only minor but very important changes in the first place.

1.1.1.2.1. MsgUpdate, MsgWait, ThreadSleep, ThreadSleepEx, ThreadDelay

All these functions expect a ***timeout*** value as one of there arguments. The traditional value for ***no timeout*** was 0. That is no longer true for mcat 2.20. Use the constant **SYS_WAIT_INFINITE** instead of 0. If you don't change this, your application will not operate! If you call one of these functions with 0, they will return **IMMEDIATELY**.

1.1.1.2.2. Includefiles

With mCAT 2.10 almost all system include files have been included via ***mCAT.h***. For some technical reasons, this is not longer true for ***memmgr.h.*** So if you need memory functions like *MemAlloc* or *MsgPoolFree* or any other, you have to add

```
#include <memmgr.h>
```

to your source.

Another include file also needs a modification. You may have used SIMPLEIO functions (such as *WrStr(), WrLn(), ...*) to print debug messages. Therefore you included

```
#include <io.h>
```

This include is no longer available. That is because its name conflicts with a standard C include file. Please change to:

```
#include <simpleio.h>
```

1.1.2. ARM7 Watchpoints

1.1.2.1. Miss Aligned Access

The ARM7 can not access long words (32-Bit) on unaligned addresses (addresses where the lower two bits have another value than 0) and it can not access short words (16-Bit) on unaligned addresses (addresses where the lowest bit is not 0).

Usually this is not a problem and the compiler will fix this problem for you. However, sometimes you may run into trouble. This happens for example when you pass a *void or byte aligned pointer* to some function and cast it then into an *struct pointer* or *a integer pointer.* The first can point to any byte on any address. The second will usually crash when the pointer is used for the first time:

```
void function_crash(INTEGER *hugo)
{
	if (hugo) *hugo++;
}
	...
```

```
      UINT8 paul[4];
      void pass;
      ...
       pass = &paul[1];
       function_crash(pass);
```

Even if this example is a bit far-fetched, it is a fact that constructions like this may occur in you source – and usually the problem is not so easy to understand as it is in our example.

Whenever an unaligned access occurs, the RAM will execute a system fault trap. This is a non recoverable situation. It will output a CPU register status and die away (executing a Sys-Reset). This can easily be demonstrated using SYSMON command ***out***:

```
2+>out long 402004 3 0

CPU FAULT: DATA ABORT [UNALIGNED ACCESS]
IN THREAD 0x005ec634 OF TASK #14
reg:  r0 =00000004 r1 =00000002 r2 =007f8cec r3 =04402003
      r4 =00000000 r5 =007f8c60 r6 =007f8c5f r7 =007f88a0
      r8 =007f8cbc r9 =00000003 r10=00000000 r11=007f88b4
      r12=00000000 sp =005ec1bc lr =00be6778 pc =00be88a0
      sr =60000010
 -- RESETING NOW!!!
```

Please note that the value for the ***pc*** *(program counter) printed is the address of the instruction that causes the fault.*

1.1.2.2. Structure Member Alignment

Because single access data fetch is by far the fastest way to transfer data on the ARM7, the compiler will try to align members in structures in a way that it can fetch 32-bit and 16-bit values with a single instruction. The penalty for a fetch of a 32-Bit value from an unaligned address is high:

32-Bit fetch, aligned:

```
      ldr    %r3,[%r8]           ; one bus cycle to read data
```

32-Bit fetch, unaligned:

```
      ldrb   %r3,[%r8, #3]
      ldrb   %lr,[%r8, #2]
      orr    %r3,%lr,%r3,lsl #8
      ldrb   %lr,[%r8, #1]
      orr    %r3,%lr,%r3,lsl #8
      ldrb   %lr,[%r8, #0]
```

```
        orr    %r3,%lr,%r3,lsl #8
```

So the compiler is wise to do preventive unaligned fetches. However, sometimes it is necessary to have packed structures. The only problem is that there is no standard way to tell a compiler to pack a structure. Therefore, if you have to pack a structure use the following macro:

```
typedef PACKED_STRUCTURE struct {
        ... some definition
} GNU_PACKED_STRUCTURE <struct-name>;
```

The structure alignment problem may occur if you have messages defined that are derivated from mCAT messages (MSG). The mCAT message itself is packed but its length is not aligned to 32-Bit. Usually this is no problem as long as you do not rely on a specific message memory layout. But if you run into problems, you must use the macros ***PACKED_STRUCTURE*** and ***GNU_PACKED_STRUCTURE.***

1.1.3. What if I Want to Maintain my Application with both the 2.10 and the 2.20?

Keep your existing mCAT2.10 makefile by renaming it to make210.mak. Insert the following line into this makefile

```
CMD=-DSYS_WAIT_INFINITE=0
```

You may already have a CMD line:

```
CMD=-D$(CORE)
```

In this case append the above separated by a space:

```
CMD=-D$(CORE) -DSYS_WAIT_INFINITE=0
```

If you use packed structures, you have to add:

```
CMD=-D$(CORE) -DSYS_WAIT_INFINITE=0 -DPACKED_STRUCTURE -DGNU_PACKED_STRUCTURE
```

You can execute the new makefile with the option -f of vmake:

```
vmake -f  make210.mak
```

Your C-Source needs no change.

1.1.4. If it doesn't Work!

MCAT is an elaborated software system. Usually, your application should be compilable after those changes. However, if you got any problems to get your existing code compiled, please do not hesitate to send your source code and makefile(s) to support@mocom-software.de.

2. Hardware Related Information

2.1. General Information

2.1.1. USRLED and BITBUS activity LED

If the hardware supports at least one software controlable LED, mCAT „flickers“ this LED at about 0.6 sec. By use of USRLED call the user can take control of this LED for private use. MCAT will not touch this LED while it is under user control. This function can be used to controll additional LEDs, too.

void USRLED(short usr, short on)

Parameter: ***if „usr“ is 0, the LED is set under system control.***

If “usr” is set to 1, the LED is under user control.

Any other value for 'usr' can be used to control additional LEDs.

If „on“ is TRUE (not 0), the LED is switched ON else OFF.

Include: ***<ticker.h>***

Examples:

```
USRLED(1,0);          // set user control, LED OFF
USRLED(1,1);          // set user control, LED ON
USRLED(0,0);          // pass control back to system
USRLED(1,1);          // switch additional LED ON, system LED is not affected.
```

2.1.2. Boot Control

To force boot mode (= to start BOOTMON instead of mCAT), usually a jumper or „magic“ switch can be used. This is different for all hardware devices and must be checked!

2.2. Hardware Reference

2.2.1. NET-A7

2.2.1.1. CPU + mCAT CORE/HARD Macro

CPU=S3C4530 // SAMSUNG ARM7

CORE=S3C4530

HARD=NETA7

2.2.1.2. Memory

Memory	***Access***	***Type***	***Location***	***User***
EPROM/ BOOTFLASH	16-Bit	FLASH	000000h-3FFFFh	-
FLASH	16-Bit	AM29F160	040000h-1FFFFFh	040000h-1DFFFFh
RAM	16-Bit	512k	800000h-87FFFFh	802000h-HEAP-START*
EEPROM	I²C, 16-Bit	ST24C64	0..1fffh	40..1EFF

Table 22: NETA7 Memory Layout

2.2.1.3. USRLED

The green Led is used as USRLED (system LED).

The yellow Led signals ETHERNET traffic.

The red Led can be switched under user control using USRLED(2,<on>). Please note that switching this ALARM-LED does not influence the function of the system LED.

2.2.1.4. BOOT Control

The TSMCPUARM is forced into BOOTMODE using the BOOTJUMPER J1

2.2.2. TSMARMCPU

2.2.2.1. CPU + mCAT CORE/HARD Macro

CPU=S3C4530 // SAMSUNG ARM7

CORE=S3C4530

HARD=TSMARMCPU

2.2.2.2. Memory

Memory	***Access***	***Type***	***Location***	***User***
EPROM/ BOOTFLASH	8-Bit	EPROM/FLASH > 64k	000000h-7FFFFh	-
FLASH	32-Bit	2*AM29F160	800000h-BFFFFFh	800000h-BCFFFFh
RAM	32-Bit	Std. 4 * 512k	400000h-5FFFFFh	402000h-HEAP-START*
EEPROM	I^{2}C, 16-Bit	CAT34WC02	0..3Fh	10..7F

Table 23: TSMARMCPU Memory Layout

2.2.2.3. USRLED

The green Led is used as USRLED.

The yellow Led signals BITBUS traffic.

2.2.2.4. BOOT Control

The TSMCPUARM is forced into BOOTMODE if the TSM-Bus Terminator is unplugged. Because operation of TSM-Systems without a Terminator is prohibited, this is also a safety mechanism.

2.2.3. TSMCPUH2

2.2.3.1. CPU + mCAT CORE/HARD Macro

CPU=TMP94C251 (“H2”).

CORE=H2_COR

HARD=TSM_H2

2.2.3.2. Memory

The TSMCPUH2 is available in four versions:

- 32-Bit Version with an 8-Bit BOOT Flash, a 2MB 32-Bit FLASH and 2MB 32-Bit RAM

- 8-Bit Version with a single 8-Bit Flash and 512k 8-Bit RAM

32-Bit Configuration

Memory	***Access***	***Type***	***Location***	***User***
EPROM/ BOOTFLASH	8-Bit	EPROM/FLASH > 64k	C00000h-FFFFFFh	-
FLASH	32-Bit	2*AM29F800	800000h-9FFFFFh	800000h-9BFFFFh
RAM	32-Bit	Std. 4 * 128k	400000h-47FFFFh	402000h-HEAP-START*
		4 * 512k	400000h-5FFFFFh	402000h-HEAP-START*
EEPROM	I^{2}C, 16-Bit	CAT34WC02	0..3Fh	30-3Fh

Table 24: TSMCPU32H2 Memory Layout

8-Bit Configuration

Memory	***Access***	***Type***	***Location***	***User***
FLASH	8-Bit	AM29F040	800000h-87FFFFh	800000h-84FFFFh
RAM	8-Bit	512k	400000h-47FFFFh	402000h-HEAP-START*
EEPROM	I^{2}C, 16-Bit	CAT34WC02	0..3Fh	30-3Fh

Table 25: TSMCPU08H2 Memory Layout

* Heapstart can be read using the “MEM” command of sysmon.

As long as no special features of the current memory model are used (flash page size, size of memory), applications will not be affected by the different memory models - beside the fact that the 8-Bit model limits the speed of the CPU!

Applications developed for TSM900CPU must – at least – be recompiled!

2.2.3.3. USRLED

The green Led is used as USRLED.

The yellow Led signals BITBUS traffic.

2.2.3.4. BOOT Control

The TSMCPUH2 is forced into BOOTMODE if the TSM-Bus Terminator is unplugged. Because operation of TSM-Systems without a Terminator is prohibited, this is also a safety mechanism.

2.2.4. DINX

2.2.4.1. CPU + mCAT CORE/HARD Macro

DINX uses an TMP95C265 or TMP95FY64

TMP95FY64: CORE=FY64

TMP95C265: CORE=C265

HARD=DINX

2.2.4.2. Memory

Memory	***Access***	***Type***	***Location***	***User***
FLASH (C265)	8-Bit	AM29F800	800000h-8FFFFFh	800000h-8BFFFFh
FLASH (FY64)	16-Bit	AM29F200 comp.	FC0000h-FFFFFFh	FC0000h-FDFFFFh
RAM	8-Bit	512k	400000h-47FFFFh	402000h-HEAP-START*
EEPROM	I^2C, 16-Bit	CAT34WC02	0..3Fh	10..3Fh

Table 26: DINX Memory Layout

* Heapstart can be read using the “MEM” command of sysmon.

2.2.4.3. USRLED

The INFO Led (green) is used as USRLED.

2.2.4.4. BOOT Control

Use a pencil to press and hold „boot mode“ button. Issue a RESET command or switch power OFF/ON. Release button.

2.2.5. NET900H/H+

2.2.5.1. CPU + mCAT CORE/HARD macro

NET900H/H+ use TMP95C063.

CORE=NET_H

HARD is not defined.

2.2.5.2. Memory

Memory	***Access***	***Type***	***Location***	***User***
FLASH	16-Bit	AM29F800	800000h-8FFFFFh	800000h-8BFFFFh
RAM (H)	8-Bit	512k	400000h-47FFFFh	402000h-HEAP-START
RAM (H+)	16-Bit	1024k	400000h-4FFFFFh	402000h-HEAP-START
EEPROM	I^2C, 16-Bit	CAT34WC02	0..3Fh	30-3Fh

Table 27: NET900H Memory Layout

* Heapstart can be read using the “MEM” command of sysmon.

2.2.5.3. USRLED

No LED is provided on the NET900 modules.

2.2.5.4. BOOT Control

Jumper J20 is used for BOOT control. Setting the jumper forces BOOTMON operation.

2.2.6. ECBCPU900

2.2.6.1. CPU + mCAT CORE/HARD Macro

ECBCPU900 uses TMP94C241 (“H2”).

CORE=H2_COR

HARD=ECB_900

2.2.6.2. Memory

Memory	*Access*	*Type*	*Location*	*User*
FLASH	16-Bit	AM29F800	800000h-8FFFFFh	800000h-8BFFFFh
RAM	16-Bit	Std. 128k	400000h-41FFFFh	402000h-HEAP-START*
EEPROM	I^2C, 16-Bit	CAT34WC02	0..3Fh	30-3Fh

Table 28: ECBCPU900 Memory Layout

* Heapstart can be read using the “MEM” command of sysmon.

2.2.6.3. USRLED

The green Led is used as USRLED.

The yellow Led signals BITBUS traffic.

2.2.6.4. BOOT Control

BOOT mode is controlled by Jumper J50. Close it to force BOOTMON mode.

2.2.7. BIT900

2.2.7.1. CPU + mCAT CORE/HARD Macro

BIT900, BIT900a, BIT900-104 and BIT900-PCI use TMP95C061.

CORE=T9H_COR

HARD=BIT_900 (REV < 9850) / HARD=BIT_900A (REV >= 9850)

2.2.7.2. Memory

Memory	*Access*	*Type*	*Location*	*User*
FLASH	16-Bit	AM29F800	800000h-8FFFFFh	800000h-8BFFFFh
RAM	8-Bit	Std. 128k	400000h-41FFFFh	402000h-HEAP-START*
EEPROM	I^2C, 16-Bit	CAT34WC02	0..3Fh	30-3Fh

Table 29: BIT900 Memory Layout

* Heapstart can be read using the “MEM” command of sysmon.

2.2.7.3. USRLED

The green Led is used as USRLED.

2.2.7.4. BOOT Control

BOOT mode is controlled by a bit in the ISA/PCI interface of the BIT900. It can be set using a suitable driver.

2.2.8. TSM900

2.2.8.1. CPU + mCAT CORE/HARD Macro

TSM900 uses an TMP95C061.

CORE=T9H_COR

HARD=TSM_900

2.2.8.2. Memory

Memory	***Access***	***Type***	***Location***	***User***
FLASH	8-Bit	AM29F40	800000h-87FFFFh	800000h-85FFFFh
RAM	8-Bit	Std. 128k	400000h-41FFFFh	402000h-HEAP-START*
EEPROM	SPI, 16-Bit	93C66	0..FFh	10..FFh

Table 30: TSM900 Memory Layout

* Heapstart can be read using the “MEM” command of sysmon.

2.2.8.3. USRLED

The INFO Led (red) is used as USRLED.

2.2.8.4. BOOT Control

The TSMCPU900 is forced into BOOTMODE if the TSM-Bus Terminator is unplugged. Because operation of TSM-Systems without a Terminator is prohibited, this is also a safety mechanism.

2.3. The Valid Interrupt-ID (intid) Values

Interrupt ID	***DMA start vector***	***TMP 95C061***	***TMP 95C063***	***TMP 95-FY64***	***TMP 94C241***	***ARM7 S3C4530***
IntReset	-	X	X	X	X	
IntSwi1	-	X	X	X	X	
IntSwi2	-	X	X	X	X	
IntSwi3	-	X	X	X	X	
IntSwi4	-	X	X	X	X	
IntSwi5	-	X	X	X	X	
IntSwi6	-	X	X	X	X	
IntSwi7	-	X	X	X	X	
IntNMI	-	X	X	X	X	
IntWatchDog	-	X	X	X	X	
IntLine0	DMAV_INT0	X	X	X	X	
IntLine1	DMAV_INT1		X	X		
IntLine2	DMAV_INT2		X	X		
IntLine3	DMAV_INT3		X	X		
IntLine4	DMAV_INT4	X	X	X	X	
IntLine5	DMAV_INT5	X	X	X	X	
IntLine6	DMAV_INT6	X	X	X	X	
IntLine7	DMAV_INT7	X	X	X	X	
IntLine8	DMAV_INT8		X	X	X	
IntLine9	DMAV_INT9				X	
IntLineA	DMAV_INTA				X	
IntLineB	DMAV_INTB				X	
IntTimer0	DMAV_INTT0	X	X	X	X	
IntTimer1	DMAV_INTT1	X	X	X	X	
IntTimer2	DMAV_INTT2	X	X	X	X	
IntTimer3	DMAV_INTT3	X	X	X	X	
IntTimer4	DMAV_INTT4		X	X		
IntTimer5	DMAV_INTT5		X	X		
IntTimer6	DMAV_INTT6		X	X		
IntTimer7	DMAV_INTT7		X	X		
IntTimer4A	DMAV_INTTR4	X			X	
IntTimer4B	DMAV_INTTR5	X			X	
IntTimer5A	DMAV_INTTR6	X			X	
IntTimer5B	DMAV_INTTR7	X			X	

Interrupt ID	***DMA start vector***	***TMP 95C061***	***TMP 95C063***	***TMP 95-FY64***	***TMP 94C241***	***ARM7 S3C4530***
IntTimer6A	DMAV_INTTR8				X	
IntTimer6B	DMAV_INTTR9				X	
IntTimer7A	DMAV_INTTRA				X	
IntTimer7B	DMAV_INTTRB				X	
IntTimer8A	DMAV_INTTR8		X	X		
IntTimer8B	DMAV_INTTR9		X	X		
IntTimer9A	DMAV_INTTRA		X	X		
IntTimer9B	DMAV_INTTRB		X	X		
IntTimer8OV	DMAV_INTTO8			X		
IntTimer9OV	DMAV_INTTO9			X		
IntRx0	DMAV_INTRX0	X	X	X	X	
IntTx0	DMAV_INTTX0	X	X	X	X	
IntRx1	DMAV_INTRX1	X	X	X	X	
IntTx1	DMAV_INTTX1	X	X	X	X	
IntRx2	DMAV_INTRX2			X		
IntTx2	DMAV_INTTX2			X		
IntAD	DMAV_INTAD	X	X	X	X	
IntDma0	-	X	X	X	X	
IntDma1	-	X	X	X	X	
IntDma2	-	X	X	X	X	
IntDma3	-	X	X	X	X	
IntDma4	-				X	
IntDma5	-				X	
IntDma6	-				X	
IntDma7	-				X	

Interrupt ID	*DMA start vector*	*TMP 95C061*	*TMP 95C063*	*TMP 95-FY64*	*TMP 94C241*	*ARM7 S3C4530*
INT_LINE_0						X
INT_LINE_1						X
INT_LINE_2						X
INT_LINE_3						X
INT_UART_TX_0						X
INT_UART_RX_0						X
INT_UART_TX_1						X
INT_UART_RX_1						X
INT_GDMA_0						X
INT_GDMA_1						X
INT_TIMER_0						X
INT_TIMER_1						X
INT_HDLC_TX_0						X
INT_HDLC_RX_0						X
INT_HDLC_TX_1						X
INT_HDLC_RX_1						X
INT_ETH_DMA_TX						X
INT_ETH_DMA_RX						X
INT_ETH_MAC_TX						X
INT_ETH_MAC_RX						X
INT_I2C						X

Table 31: The Valid Interrupt-ID (intid) Values

3. BOOTMON SLDR Commands

3.1. The Serial Line LoaDeR

If BOOTMON is active, SLDR is available on ALL serial lines at 19200-8n1. Even if you can use any serial line to get in touch with SLDR, you should only use one connection at a time. If problems arise caused by other devices connected to other serial lines, simply unplug them while working with BOOTMON!

SLDR supports a small subset of SYSMON commands. It uses almost the same syntax & semantics.

3.2. Argument Format

As with SYSMON in mCAT, SLDR expects all arguments as hex values by default. A hex value must start with a digit, so you will get an error trying:

```
eewrite 0 ff
```

The correct line is

```
eewrite 0 0ff
```

If you wish to use a decimal number, please use the „#“ char in front of the decimal value:

```
eewrite 0 #255
```

All input to SYSMON and SLDR must be lower case.

3.3. SLDR Commands

eewrite

Command is used to write values to the serial EEPROM. The format is:

```
eewrite <addr> <value>
```

The address range is limited by the used EEPROM. As a common rule we expect to address 0..255 words of EEPROM. If more than one EEPROM is used in the system, the msb of <addr> is used to address the chip:

```
eewrite 102 4
```

Writes the word value „4“ to third word of chip 2!

blank

Checks a region of memory (usually within the FLASH) for programmability. If no arguments are given, blank checks the region of a FLASH that is reserved for the mCAT kernel itself. If non-blank regions are found, they are listed on the screen.

```
blank {<start-addr> <end-addr>}
```

reset

This command sets up the watchdog timer, disables all interrupts and waits for the watchdog to reset the system. By using the watchdog, a real RESET is issued, it is not only a „jump“ to the program start!

erase

Use erase to erase pages of the flash memory *immediately*. This command can be used to remove dead user programs or even to remove mCAT itself. However, with delpage we offer a more convenient command. See „delpage“.

```
      erase <addr>
```

where <*addr*>is the start address of the page to be deleted.

delpage

The delpage command only marks pages of Flash for deletion rather than issuing the real *erase* command. After marking several pages as „to be deleted“, a reset command is issued and the pages are deleted at the restart process. This procedure is recommended for some reasons:

- The *fera* module, that executes the erase process after reset, knows much more about the real FLASH used. For example in cases where the reserved mCAT region consists of multiple Flash pages (as is the case with the AMD29Fx00 type memory's) *fera* gathers all those pages and deletes them all with a single command.
- If you have a flash memory that mirrors in the address space, erase will work with the mirrored addresses, too. Delpage will not.

Delpage takes a page number as an argument. This number is counted from 0 to 15 (0f). To erase the mCAT kernel - for example - you have to issue:

delpage 6	on TSM900, BIT900
delpage 0f	on NET900H/H+, BIT900A, BIT900-104
delpage 3	on all systems using the TMP95FY64 CPU

See Appendix C for a table of Flash memories and page numbers.

purge

By using *purge* an mCAT module (including the entire mCAT kernel itself) can be marked as „not startable“. Such a module will be ignored in the mCAT startup process. This command allows the deactivation of modules that do not work and block the startup process for any reasons.

```
purge <addr>
```

The address expected by purge is the base address of the modules IMD - except for the kernel! To „purge“ the kernel, the base address of the mCAT reserved page must be given. See Appendix C2 for more information.

3.4. Downloading Motorola S3-Hexfiles

The simplest, fastest and best way to download any S3 file is using mocoms LGO.EXE terminal program for DOS or the forthcoming wLGO.EXE for WIN95/98/NT. You can specify the S3-file as a command line parameter or interactively by use of hotkey „F3“.

However, other terminal programs work, too. Set them up for 19200-8n1, select „ASCII DOWNLOAD“, set the „line pacing“ to 30ms (the time to program Flash memory) - and it should work! The „line pacing“ will add some „delay“ at the end of a line.

Please note that no command must be issued: Just start the download, SYSMON / SLDR will detect S3 records on the fly!

3.5. Replacing mCAT

If your system is running mCAT and you want to replace mCAT by a more recent version, follow 4 simple steps to replace mCAT (this is the normal mCAT replacement procedure):

1. Use the „purge“ command on the mCAT base address (see Appendix C2) from the SYSMON command line (or from BITMON if you use BITBUS) to invalidate the kernel. For example: use *purge 8f0000* for NET900H with an AM29F800 Flash.
2. Use „delpage“ to mark the mCAT page for deletion. For the NET900H use *delpage 0f*.
3. Reset the system. BOOTMON will erase the page 0f and start right away. You can issue a *blank* command to check if the page is really clean.
4. Download the new mCAT and issue a reset afterwards. Ready!

If the system hangs (for any reason), you have two alternatives:

If only the user supplied module hangs:

1. Set the BOOT MODE. See 4.4. for details on your hardware. For NET900H, simply close JUMPER 20.
2. Reset the system by switching OFF/ON.
3. Now you are under BOOTMON control. Remove the BOOT MODE plug/jumper NOW!
4. Use the purge command to disable all „mad“ modules.
5. With a reset you get back to mCAT - if you have not forgotten to unplug the BOOT MODE jumper/plug (step 3)!

If the system hangs because the download of the mCAT kernel failed or for any reason mCAT got damaged, follow this list:

1. Set the BOOT MODE. See 4.4. for details on your hardware. For NET900H, simply close JUMPER 20.
2. Reset the system by switching OFF/ON.
3. Now you are under BOOTMON control. Remove the BOOT MODE plug/jumper NOW!
4. Use purge to disable all modules including mCAT! Issue a delpage command to erase mCAT (and other pages if you like).
5. Continue with step „3“ of the normal mCAT replacement procedure

4. Supported Flash Types

Manufacturer	***128kByte***	***256kByte***	***512kByte***	***1024kByte***
AMD	AM29F010 AM29F100T	AM29F200T	AM29F040 AM29F400T	AM29F800T
TI	AM29F010	-	AM29F040	-
TOSHIBA	-	TMP95FY64	-	-
FUJITSU	MBM29F010	MBM29F200T	MBM29F040 MBM29F400T	MBM29F800T
ST	STM29F100T	STM29F200T	STM29F040 STM29F400T	-
HYUNDAI	-	HY29F200T	HY29F040 HY29F400T	HY29F800T

Table 32: Supported Flash Memories

A. Index