DEPENDABLE SOFTWARE (BOSS) FOR THE BEESAT PICO SATELLITE

Sergio Montenegro(1), Klaus Briess(2), Hakan Kayal(2)

(1) FhG FIRST; Kekulestr 7; 12489 Berlin; sergio@first.fhg.de
(2) Technical University Berlin; Marchstr 12; 10587 Berlin; {Klaus.briess, hakan.kayal}@ilr.tu-berlin.de

1. ABSTRACT

BEESat (fig. 1) is a picosatellite project (CubeSat standard) under development at the Institute of Aeronautics and Astronautics of TUB (Technical University of Berlin), Berlin, Germany, in cooperation with AstroFein technik (Berlin) and the Fraunhofer institute FIRST .

All satellite functions are controlled by software running in the dual redundant bord computer. This control software has to provide a very high dependability label. We selected the BIRD Operating System BOSS as basic framework to build all applications on the top of it as building blocks. BOSS (from FhG FIRST) is a real-time embedded operating system and middleware, which were designed for safety and simplicity. Its main design goal was dependability.

2. THE BEESAT PROJECT

The overall objective is to demonstrate and verify new component technologies for picosatellites, in particular the qualification of microwheels in orbit - an enabling actuation device to improve the attitude control capabilities to a new functional level in the very restricted confines of a CubeSat.

Figure 1: BEESAT

Although there are currently many developers of CUBESAT's, many of them lack of a precise attitude control system. This in turn is mandatory for more sophisticated applications, which require pointing capability. Having a precise attitude control system opens the door to new potential and very cost effective applications, which includes areas such as earth observation, space science, astronomy and on-orbit verification of new technologies.

The combination of highly sophisticated miniaturised technologies and the advantage of building swarms or clusters of pico satellites will result in completely new applications at very low cost. Beesat will demonstrate the use of coin size micro wheels (Fig. 2) for the attitude control of pico satellites in orbit as one of the key elements on which TU-Berlin is working. Further missions with a focus on other key technologies are planed for the future.

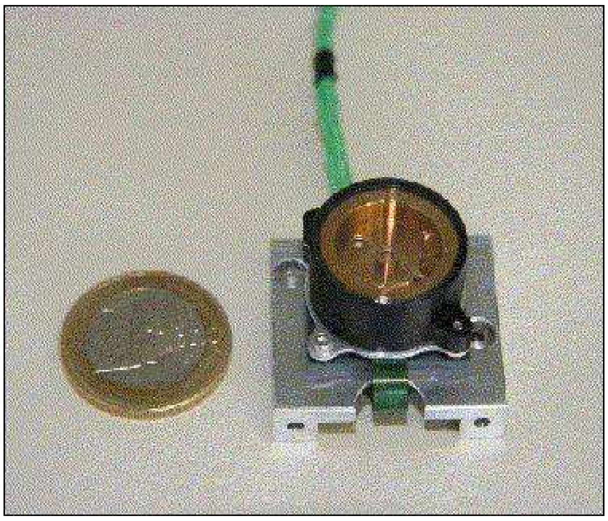

Figure 2: microwheel compared with a 1 EURO coin

The main design requirements of BEESat can be summarized as follows:

- LEO orbit in the range 450 - 850 km
- Lifetime of 1 year
- 3-axis stabilization
- RF communications in UHF band
- Spacecraft operations at TUB

The launch of BEESat is planned for the year 2007.

3. SPACECRAFT

BEESat conforms to the CubeSat standard having a size of 10 cm x 10 cm x 10 cm, and a mass limit of 1 kg, (fig. 1) permitting a launch with the P-POD deployment mechanism of CalPoly. The spacecraft is 3-axis stabilized utilizing "microwheels" designed and developed by TUB in cooperation with Astro- und Feinwerktechnik Adlershof GmbH (Astrofein).

Each microwheel (fig. 2) is having an angular momentum of at least 2 x 10-4 Nms with a minimum torque of 3 x 10-6 Nm; the size of a single device is 20 mm in diameter and 14 mm in height. Three or four of the microwheels are suitable for an actuator system of a picosatellite providing a 3-axis stabilization capability. The microwheels are being controlled with a common electronics board which is connected to the spacecraft bus via a CAN (Controller Area Network) interface.

The BEESat ADCS (Attitude Determination and Control Subsystem) utilizes 3 microwheels for 3-axis stabilization. In addition, a magnetic coil system is being used to desaturate the wheels and to control the satellite during orbital periods when the wheels might not be available. Attitude sensing is provided by a sun sensor system based on 6 PSD (Position-Sensitive Detector) photocells of Hamamatsu, and a magnetometer of Honeywell (HMC1023). The ADCS software supports 4 main attitude modes: a) inertial pointing, b) maximum power, c) Earth pointing, and d) rotation mode (as well as intermediate modes).

The OBDH (Onboard Data Handling Subsystem) utilizes a microcontroller (Philips LPC2292), a flash memory for data storage, and a redundant CAN bus interface for the communication with the microwheels. The OBDH employs the TinyBOSS operating system of FIRST (Fraunhofer Institute of Computer Architecture and Software Technology), Berlin. TinyBOSS is an adapted version of BOSS, a real-time operating system being successfully flown on the BIRD (Bi-Spectral Infrared Detection) microsatellite mission of DLR. Note: the acronym BOSS stood originally for "BIRD Operating System (Simple)" - it is now being used for many other applications, not only for those in space.

Power is being provided by surface-mounted solar cells. Four Li-ion battery cells are being used for spacecraft operations during the ecliptic phase of the orbit. The power bus provides a voltage of 3.3 V and 5 V to the S/C subsystems, utilizing DC/DC converters.

4. THE OPERATING SYSTEM BOSS AND ITS MIDDLEWARE

BOSS (from FhG FIRST) is a real-time embedded operating system and middleware, which were designed for safety and simplicity and to allow their own mathematically formal verification. Complexity is the root of most development errors – if you eliminate complexity, you eliminate most development errors. This was one target of BOSS. The BOSS-middleware simplicity allows the system to be easily understood, used and ported to other platforms even to FPGA (Field Programmable Gate Array) logic. The BOSS middleware has already been implemented and is being used in software for real-time dependable systems. We now aim to implement the same middleware in FPGA hardware. This will bring hardware and software developers together in an environment that is familiar to both. It makes no difference whether applications running on top of the BOSS middleware are implemented in software or hardware. For communication purposes, it makes no difference whether the communication partner is implemented in hardware, software or both. The BOSS middleware allows any combination of communication (SW/SW, SW/HW, HW/SW, HW/HW) and creates an internal satellite standard interface between software and hardware without needing different device drivers for different devices.

4.1. BOSS Middleware

The BOSS-Middleware was designed to support fault tolerance. All processes running on top of the BOSS-Middleware can exchange messages asynchronously using a subscriber protocol: a process or a hardware device can subscribe to one or more message types by name. When a process or a hardware device sends a message of a given type (name), each subscriber to this name receives a copy of the message. For communication purposes, the node and even the software/hardware barriers/boundaries are transparent. The messages are distributed across these barriers. Using this approach, we obtain very high flexibility and users do not have to differentiate between local/remote functions or hardware and software functionality. The system can be configured or reconfigured simply by plugging software modules or hardware devices into/out of the middleware.

The BOSS-Middleware provides transparent support for fault tolerance. The simplest example of this is a controller sending commands (messages) to a device. (Fig. 3) As a first step, we insert the middleware between the device and the controller by implementing the same interface on both sides of it (Fig. 4). Neither the controller nor the device notices this intervention. The middleware forwards the messages across node boundaries, which means that controller and device no longer need to be located in the same node. Furthermore, messages can be replicated if there is more than one subscriber to a message type (name). Now we can add a monitor to hear messages of the same type, like the device. The monitor can create a log file and/or execute an online diagnosis of the system. Again, no one will notice this intervention.

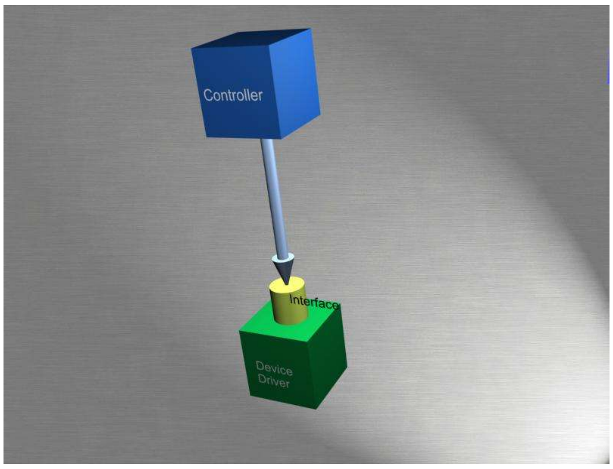

Figure 3: Controller-device, No middleware

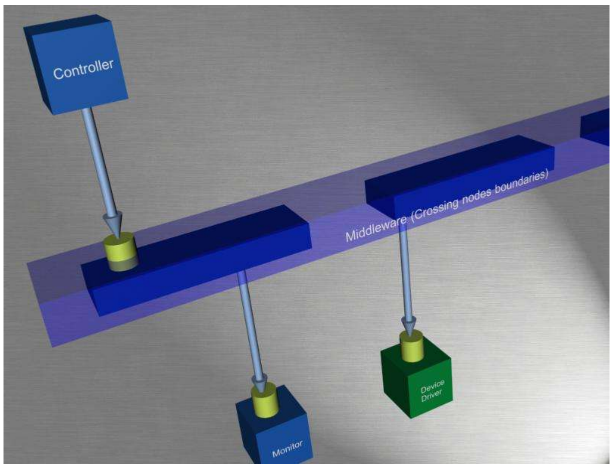

Figure 4: middleware insertion

The next step is to replicate the controller, simply by creating several instances of it, if possible running on different nodes (Fig. 5). They need not know about the existence of the other replicas. What are needed now are voters that intercept all messages to the device, compare them and send only those that are most likely to be right (a democratic decision, e.g. two of three) to the device. If required, it is possible to replicate the voter, too. One voter – the worker (as in BIRD) – is in charge and the other one – the supervisor (as in BIRD) – is a hot redundancy. The supervisor is ready to take control if the voter in charge fails to respond .

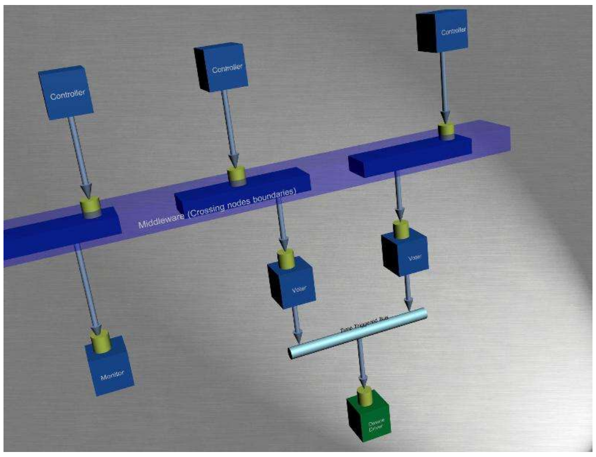

Figure 5: middleware handling service replication

The routing of messages depends only on the types/names of the messages and on who is subscribed to each name.

4.2. MESSAGE DISTRIBUTION

The BOSS Middleware replicates and distributes messages asynchronously using a publisher/subscriber protocol. Tasks (Threads too) which provide services publish each of them under a given name (sending messages with a name). Any task can subscribe to one or more services (message types by name). The middleware keeps a list of subscriber names each with a reference to the subscriber object (subclass of "Message")

The user creates own messages by extending (inheritance) the class "**Message**". Message provides a method "send(char *name)" to send its own data. The name string of "send()" identifies the service name or (equivalent) the messages type. "send()" access the "**NameServer**" built-in in the middleware, in order to find all subscribers to the given name. The "NameServer" has, for each registered name, a reference to the associated receiver object. In this case a subclass of "Message". Each name can be repeated with different associated objects. This would mean the named service has more than one subscriber which is a normal case.

Using the built-in iterator, the "NameServer" goes through the name list, and for each matching entry, it calls the methods "copyDataFrom(Message *from)" and "execute(char *name)" of the associated object (subclass of "Message"). Thusly each subscribed message gets the change to get (or discard) a copy of the transmitted data (using "copyDataFrom()") and then to process it (using "execute()").

If the sent message has to be forwarded across an external network, like for example Ethernet, CAN-Bus or a serial link, the middleware calls the method "size()" of the calling object (a subclass of "Message"). The byte count returned by "size()" is used to serialise and send the message through the network.

If required, each node has a middleware network server which gets messages from the network and distributes them using the local middleware like any other local task.

5. BOSS KERNEL

The kernel performs local resource administration like thread scheduling, time management and interrupt propagation; and provides some basic services for thread synchronisation and basic local communication.

The user shall extend the BOSS kernel framework by means of inheritance from the "**Thread**" class, in order to implement his functionality or application . The subclasses of "Thread" are integrated automatically in the framework and can be administrated by the kernel.

The kernel keeps a list of all user threads (active objects, subclasses from "Thread") and activates each of them at the appropriated moment.

Inside of the kernel there are three objects which are active by themselves without requiring user subclasses. These three objects manage user threads and resources.

1. The **Time manager** administrates the local time at each node. Time is a microseconds counter stored in 64-Bit variables. Time 0 is the boot time of the node. "ENDOFTIME" is a constant containing the highest possible value for 64 Bits. It correspond to approximated 300 000 years in the future. TimeManager also provides conversion from/to micro seconds counter to/from UTC and Gregorian-date.

2. The **Scheduler** keeps a list of all threads (static array inside of Thread) and according to priority, time to start and block, it assigns CPU time (this means activation) to each of them.

3. The **Interrupt manager** keeps a list of user interrupt servers. Each time an event or interrupt occurs the InterruptManager searchs its list of servers and calls the method "eventServer()" of the corresponding (registered) Thread.

The classes to generate passive objects implement help functionality to support the teamwork between active objects. These functions are:
for synchronisation ("TimeController", "Semaphore", "SignalBox")
data sharing/passing ("AsyncComFifo", "AsyncComBuf") and
combined data sharing/passing and synchronisation ("MailBox").

6. BOSS HARDWARE DEPENDENCIES

The hardware dependent layer is very small and implements the functionality which is different on each platform, e.g. CPU-register load/store, low level Hardware-Drivers and basic interrupt management. This is the only construction site in order to be able to move from one platform to another. The higher layers do not need any modification.

The interaction between the hardware dependent layer (**HDL**) and the rest of the system is shown in figure 6.1

The kernel invokes services form the HDL using normal functions calls (no object invocations). Such functions are for example to perform a context switch from one task to another (transfer), to turn interrupts on/off, node reset, atomar operations, get/set time and set timer interrupts, etc. The kernel and higher layers decide when to perform this activities. The HDL transforms these calls in Hardware register access, going directly to the CPU, timers and HW registers or executing assembler instructions which are not available in higher languages.

The HDL reacts to interrupts from the Hardware, by invoking the basic interrupt server like a function call (call from interrupt). The basic interrupt server will firstly access some hardware registers to get more information about the interrupt and reset it, then if required it performs an upcall to the Kernel, which can propagate the upcall to higher layers. Note: the upcalls are executed asynchronously from all other thread functions and on the stack of the current thread. Therefor by implementing a user interrupt server (upcall) care has to be paid to avoid data corruption. Semaphores will not work at this level.

Higher layers, HDL and hardware can access common memory locations to communicate. Because upcalls work asynchronously to threads and no semaphore protection can be provided, the communication shall relay on atomar variables (simple types like integer and character) read/write (with ony one reader and only one writer) and using asynchronous communication objects for more complex data structures. For this purpose the user can create objects from "AsyncComBuf" and "AsyncComFifo". This classes are implemented in such a manner, that both sides can be accessed asynchronously from each other (but only one reader and only one writer). No explicit locks or synchronisation is required.

Interrupts have the highest priority in the system. When designing the communication structure between interrupts and threads consider that a thread can be interrupted by a hardware interrupt at any time (except if interrupts are disabled), but an interrupt server will never be interrupted by a thread. If dispatching is enabled, it is quite possible that a thread switch will take place after the interrupt server returns.

For some very exotic cases it could be required to turn interrupts off. In such cases the user can turn interrupts on/off using the constructors "ATOMAR" and "ATOMAREND"but this shall be only as the last alternative if nothing else helps. Between "ATOMAR" and "ATOMAREND" interrupts are disabled, therefor this block shall be as short as possible (no longer than 3 microseconds please). "ATOMAR" and "ATOMAREND" can be nested. Internally there is a counter which counts the entries (++) and the exits (--) of this blocks. Only when the counter reaches 0 again, the kernel enables interrupts again.

4. REFERENCES

2005: Kayal, H: BEESat internet presentations:
http://directory.eoportal.org/pres_BEESatBerlinExperimentalEducationalSatellite.html
http://www.beesat.de

1999: Montenegro, S.: Entwicklung sicherheitsrelevanter Systeme, Hanser Verlag, ISBN: 3-446-21235-3

2003: Briess, K., Baerwald, W., Gill, E., Halle, W., Kayal, H., Montenbruck, O., Montenegro, S., Skrbek, W., Studemund, H., Terzibaschian, T., Venus, H.:
Technology Demonstration by the BIRD Mission
4th IAA Symposium on Small Satellites for Earth Observation, April 7-11, 2003
ISBN 3-89685-569-7

2002: Briess, K., Bärwald, W., Hartmann, M., Kayal, F., Krug, H.3, Lorenz, E., Lura, F., Maibaum, O., Montenegro, S., Oertel, D., Röser, H.P., Schlotzhauer, G., Schwarz, J., Studemund, H., Turner, P., Zhukov, B.:
Orbit Experience and First Results of the BIRD Mission
53rd International Astronautical Congress The World Space Congress 2002, October 10-19, 2002, Houston, Texas, USA

2002: Briess, K., Montenegro, S., Bärwald, W., Halle, W., Kayal, H., Lorenz, E., Skrbek, W., Studemund, H., Terzibaschian, T., Walter, I.:
Demonstration of Small Satellite Technologies by the BIRD Mission
16th Annual AIAAA/USU Conference on Small Satellites, Logan, Utah, USA 2002

2002: Montenegro, S., Barr, V.:
BOSS/Ada: An Open Source Ada 95 Safety Kit
Ada Deutschland Conference 2002, March 6-8, 2002, Jena, Germany