by Harold Price NK6K and Robert McGwier N4HY

Abstract

The evolving structure of the Microsat system software is discussed. With a launch services agreement in hand, several "Pacsats" should be in orbit in 1989; they will have more memory and a more complex suite of system and application programs than any amateur spacecraft launched to date.

Background

"Pacsat", a dedicated packet radio satellite, is a concept that has been floating around in the amateur digital and space communities for more than six years. The basic concept is simple, using a low orbit satellite to "carry" messages from party to party rather than doing a real-time transfer though a high orbit satellite that both parties can see at the same time. Microsat is a specific AMSAT-NA satellite system that hosts a Pacsat mission.

A paper in these proceedings by Tom Clark gives an overview of the current Microsat design. Another paper by Lyle Johnson describes the hardware that supports the software described in this paper.

One of the large benefits of the amateur radio "Pacsat" concept has always been low cost. Not that hams are necessarily frugal, we'd spend it if we had it. If fact, it has been said that a ham will spare no expense in buying other parts to go with something he got for free. But we haven't got it, so we're forced to be smart. There are several cost reducing concepts in Microsat. One is small size. This reduces building costs (we're talking small, not miniaturized, which drives costs up). It also reduces launch costs, which are typically pegged to launch weight. Another cost reducer is dependence on off-the-shelf commercial grade parts. We're using extended temperature range parts, for the most part, but avoiding the much more expensive space rated parts. A combination of knowing the limitations, a fairly benign low orbit,, and paying attention to the fine details of thermal control, allows us to get away less expensive electronics.

Another cost reducing element is the use of off-the-shelf software, and a low cost software development environment. As anyone who uses computers knows, a low cost software development environment has two components, the actual cost of the software itself, and the cost of the labor required to use the software to do something useful. With Microsat, a serendipitous turn of events has led to a development environment which should be both.

Off the Shelf

Microsat is a very low cost spacecraft. One reason is that it uses off-the-shelf parts and the simplest possible hardware design. The spacecraft itself is minimal, it is "stabilized" only by a bar rnagnet, which locks with the Earth's magnetic field and keeps the spacecraft from presenting the same face to the sun all the time. The interconnecting wiring harness is minimal, using a mere 25 wires and depending on serial bus controllers in each module to multiplex several control and sensor signals on a few wires. The CPU hardware uses standard commercial parts, such as a V40 CPU, closely related to the CPU found in IBM PCs, rather than some exotic, rad hard bit slicing wiz bang.

We'd also like the software to be off the shelf and straight-forward, able to take advantage of standard development tools and languages, and to use large blocks of existing protocol and bulletin board code.

With these goals in mind, Microsat will be the first amateur spacecraft to use an off- the-shelf high level language as its major implementation standard, Microsoft C. It will also use an off-the-s'helf multi-tasking operating system from Quadron Service Corporation called qCF. It should be pointed out that the University of Surrey's UoSat-D, to be launched on the same mission, will also use some of the same software, so the distinction is a shared one.

Why C?

Why, in fact, a high level language? The purpose of the Microsat spacecraft is to support a 16 bit, 5 MHz computer with 10 Mb of memory. In addition to controlling the spacecraft, the computer will support one or more complex data communication protocols, a file system, memory error detection and correction, and data compression. It will compute its sub-satellite point (the point on the ground immediately below the satellite:) so that it can switch the downlink transmitter to low power while over dead zones, or determine when to grab an image from the CCD camera. This will require floating point math. Last but not least, the software must be written in time to support a possible January 1989 launch. Doing all this in assembler would be a daunting task.

C, while not a perfect language, is at least ubiquitous. This means that many people write in it, including many hams. Microsoft C is well known, has copious documentation, has many debugging tools, and is easily available. While it has some bugs, they are well documented, and are in areas we are unlikely to use:, such as graphics. It will generate code that is small enough to fit in the available space, and will run fast enough to support any modem technology we are likely to fly for some tirne to come.

For another advantage, implementations of the amateur radio packet protocols are available in C, including the KA9Q AX.25 and TCP/IP.

C is even nicer though when a support library of interface routines are available to support inter-task communication in a real-time system such as Microsat.

Why Multi-tasking?

Figure 1 shows a number of the tasks that the Microsat CPU will have to support, organized by major function. These tasks will be described in detail later. The chart is greatly simplified, as it leaves out such things as the software to manipulate the 10 Mb of memory, 2 Mb bank switched and 8 Mb as a randomly addressable serial buffer. Many non-related tasks must be carried on concurrently, such as searching the BBS message base for all messages to W1AW while monitoring battery voltage. Some things occur on demand, such as downloading a file, others happen based on time, such as including a telemetry frame in the downlink stream once every n seconds.

One way in which this can be done is to write each function as a subroutine to a single large program and link them all together. The main program would contain a table of subroutines to call when a timer it maintains expires. It would also contain what is commonly called a "commutator loop"; a set of subroutines representing major functions which are called in turn, each does some processing and returns to the main loop. A good example of this type of program is the TCP/IP package by Phil Karn, KA9Q, and others. FO-12 also uses a commutator loop.

Commutator loops have several disadvantages, however. The program is linked into one big unit, meaning that all parts of the program have to be aware of globals and subroutine names, even parts who's only point in common is that they are both called by the same main program Because the program is one linked unit, individual parts can't be easily replaced or reloaded. While commutator-loop programs are effective in the right circumstances, in general the larger they get or the more disparate the parts, the harder they are to develop and maintain. KA9Q is also leaving the commutator behind for a home grown kernal for net.exe for reasons like those given here.

Another alternative to true multi-tasking is a multi-threaded FORTH. The Phase 3 satellites Oscar 10 and 13 use IPS, a FORTH-like system. Much has been said about FORTH and its applicability to real-time control programming, it was originally developed to control large telescopes. With FORTH you either love it or hate it, and most of the prospective programmers for the Microsat project were in the latter category. FORTH is not particularly optimized for things like AX.25 protocols and BBSes, and there are no existing amateur packet radio applications in this language. If FORTH were chosen, we'd be starting far down on the power curve.

A true multi-tasking system would have the following advantages:

1) All programs (tasks) could be writ ten as separate entities. This eliminates global name and other problems, and makes it easier for several widely separated programming groups to contribute for the effort.

2) Since tasks are separate programs, they are easier to generate and test.

3) Tasks can be easily removed and reloaded.

4) The multi-tasking scheduler ensures that all tasks get a chance to use the CPU. More types of programming errors or failures are survivable, a glitch in one task will not necessarily halt processing in other tasks.

A true multi-tasking system seems more desirable, the problem is finding one for a particular hardware system. Fortunately, serendipity smiled on us. Early this year the generic Pacsat CPU design was upgraded from the 1984-style NSC800 Z-80 8 bit class to the 16 bit 8086 class. The serial communications chip was an 8030/8530 surrogate. It just so happens that someone with a long-time interest in PACSAT software (NK6K) was one of the founders of a company that has been marketing a real-time multi-tasking communication package for an 80186/8030 coprocessor card since 1986. One of the other founders, Wally Linstruth, WA6JPR was a charter member of the ARRL Digital Committee and was one of the first packet users in southern California. In short order, the company, Quadron Service Corporation, agreed to port its software to Microsat, and to give AMSAT free development software and no-cost license agreements for the use of the system on amateur radio spacecraft.

Attributes of the Quadron Multi-tasking system

1) Pre-emptive scheduler. This is a buzzword that means a task is given a certain amount of time to run, and is then placed at the end of a list of tasks waiting to run.

2) Sleep. Tasks can put themselves to sleep, meaning that they have nothing to do at the moment, and don't need to be scheduled on the CPU. Going to sleep is not something that a normal, single task program such as any standard PC-DOS program needs to do. Since there is only one task in such a system, if it has nothing to do it might as well loop. In a multi-tasking system, other tasks may have something to do, and it is more efficient if an idle task is simply not run rather than have it loop until it gets pre-empted. Tasks are automatically put to sleep if they start an operation that can't yet be completed, for example reading an input packet when one hasn't been recieved. The scheduler will start running the task again when input is available.

3) Timers. A sleeping task can also be awoken when a timer it has started goes off.

4) Inter-task messaging. Many of the tasks in Microsat will want to exchange data with other tasks, for example, the BBS will want to send data to the AX.25 output task. This is handled through a mechanism called streams.

A stream is like a little Local Area Network (LAN) connecting tasks together. A task opens to a stream which establishes a name on the "LAN". Other tasks can send messages to that name. Messages are queued. A sleeping task will wake up when a message is sent to it. For example, the AX.25 task sleeps until a task writes a message containing outbound data to it. The AX.25 task processes this data into an AX.25 frame and then writes that frame in a message to the HDLC Driver. The AX.25 handler is also awoken when the HDLC driver sends it a received

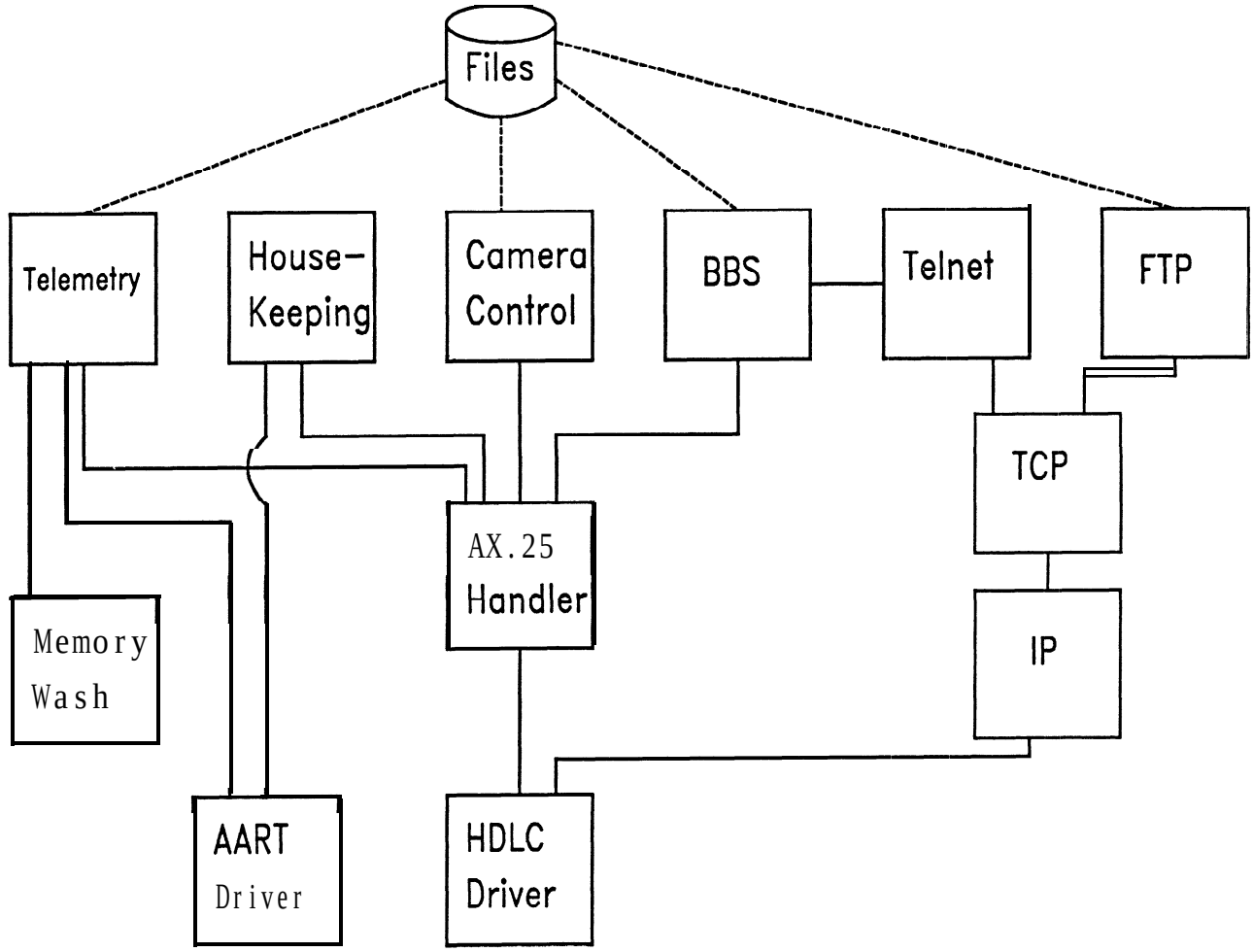

Figure 1. Possible Microsat Task Configuration

irame through a stream. Flow control limits may be set on streams, a task can. be put to sleep if it writes too many messages; it is awoken when the target task reads the messages. This keeps one task from using all the buffers by flooding a second task with messages.

The inter-task messaging is implemented with calls similar to the standard C open, close, read, and write subroutines. Aside from using these calls to send data to other tasks, and the desire to find good places to go to sleep, writing a Microsat application program such as the BBS is pretty much like writing any other C program. Large portions of these programs can be tested on a regular IBM PC using Codeview, which should considerably speed development .

Specifics

Now that we've discussed some of the major design decisions, let's review the major tasks, as shown in figure one. Most of the fine details are still being worked out, the following discussions hit the high points.

Kernal

Not shown in figure one, the kernal supplies the basic multitasking services. It manages the hardware timers, sets up memory, loads and unloads tasks.

File Support

Not shown in figure one, the file support is implemented as a task. The low level C read and write subroutines in the standard C library are replaced by routines that format an I/O request and send it through a stream to the file support task:. Acting much like an IBM PC ram disk driver, the file support task uses the 10 Mb of memory to emulate a disk. File support provides blocking and deblocking services as well as providing error correction for single bit errors (see the Mem Wash description).

BBS

This will be the most visible program to users on the ground. The major goal of the two PACSAT Microsats is to provide a bulletin board and file service. There will probably be two distinct interfaces. One will be an interface familiar to most packet users, the standard RLI/MBL BBS. This is for casual or occasional users who just want to see what's going on or forward an occasional message.

A second interface will be optimized for computer to computer transfers. While the RLI/MBL interface is currently used for this as well, Microsat will be in a different environment. The current auto-forwarding software is

used a network where several hundred stations bang away at each other 24 hours a day, or at least several hours, HF conditions permitting. In the lower latitudes, a Microsat will be visible about 10 minutes at a time, four or five times a day. We'll want to take advantage of every second of that time. We won't want to wait while messages are sent one at a time, reprompting for each new message. We won't want to discard a long message just because the satellite went out of range before the last packet was sent. We'll probably want to block a large number of messages in a single file and send full speed, letting Microsat unblock them later. If a file is partially received, we'd like to be able to continue from the last byte received on the next pass.

The second access method could be used by a smaller number of backbone forwarding stations. The Microsats will be an experimental platform for testing various ways of simultaneously maximizing message throughput and maximizing the number of users who directly interact with the spacecraft. On the surface these items appear to be mutually exclusive.

AX.25 handler

If launch occurs as scheduled, the only protocol supported will be AX.25. The software will be a derivative of the KA9Q AX.25 code. It will support a large number of simultaneous connections through all of the uplink channels. When no frames are queued for the downlink, the AX.25 handler will send a message to the Telemetry task asking it to downlink a telemetry record. The telemetry task will also periodically send data based on a timer.

Sometime after launch, (time permitting before), we should also be able to test TCP/IP as an access method in addition to AX.25. AX.25 will at all times remain available.

HDLC driver

The HDLC driver passes frames between other tasks and the uplinks and downlink. The driver is non-trivial. The hardware design supplies several DMA channels, but even so there are more I/O channels than DMA, so the driver must do both DMA and straight interrupt driven I/O. To get the most out of the available processor power, and to enable later Microsat missions to use even higher baud rates, the HDLC driver is written in assembler code. Skip Hansen, WB6YMH, is a real wizard at this sort of thing, and will be porting the HDLC drivers he wrote for Quadron to the Microsat environment.

Although the HDLC driver will probably just be feeding the AX.25 handler at launch, it will later also be the front end for the IP module.

Housekeeping

For all these features to be usable, the spacecraft must be maintained in good operating condition. For example, if the battery voltage goes below a certain preset threshold, low power only should be allowed from the transmitter until recharge. Command stations must be able to talk to the algorithms that monitor optimal settings for power, solar panel operating points, and upload targets for these control algorithms. The command must also be able to manually intervene when the automatic algorithms don't do all that we have asked them to. It might be best to a only allow low power mode over sparesely populated areas on the globe unless a valid packet is heard. These and other spacecraft maintenance functions are handled by the housekeeping modules.

Telemetry

The Telemetry module periodically gathers telemetry data by using the AART driver to collect data from other modules. The data is both sent to the downlink as a UI frame for real-time monitoring, and is also stored in a virtual disk file in memory. The "whole orbit data" format, where the values for telemetry channels are store over several hours and are later downlinked has been proved popular with users by the UO-9 and UO-11 spacecraft. This data would be available in a file and could be downloaded by command stations and users.

The Telemetry module would be addressable via the AX.25 uplink by command stations for the purpose of modifying the interval used for dumping UI telemetry frames and for storing whole orbit data. When diagnosing a problem, the sample rate would be increased, as would the total' memory to be dedicated to storing data. For example, the battery voltage can be sampled once every two milliseconds for a 24 hour period, and the data stored in 8 Mb of memory. It would, of course, take a long time to download that file.

Memory Wash

Some of the memory on the spacecraft is protected with hardware Error Detection and Correction (EDAC) circuitry. When an error is induced in memory by an energetic particle normally filtered out by the atmosphere, the EDAC will correct the error on a read and place the proper data on the bus. The corrected byte is not written back into memory automatically by the hardware. If an error is allowed to linger, there is a chance that a second bit in the same byte will get flipped. Since the hardware can only properly fix single bit errors, you'd like to fix all single bit errors before they become multi-bit. In a process called "washing memory", a task periodically runs through the EDAC memory, reading and writing every byte, causing the corrected byte to be written back into memory over a damaged one.

Most of the memory is not protected by hardware. The reason is economics, 12 bits are used to store each 8 bit byte in hardware protected memory. Hardware EDAC is used for memory that programs run out of, since a program byte in error will usually lead to no good. Software algorithms must be used to protect the remaining memory. This memory is used to store data files and messages. The ramdisk routines will use software EDAC to correct errors, but if a "disk sector" goes unread for too long, multiple bit errors may occur. To reduce this chance, the memory wash task periodically reads all "disk sectors" and writes them out.

The Memory Wash program responds to queries from the Telemetry task and reports the numbers of errors corrected and the current position in the wash cycle. This information then is incorporated into the standard teleme-

try frame.

AART Driver

This module reads AART commands from other tasks out of the message stream, and sends them to the AART serial control channel. If required, it uses the CPU boards A to D converter to read a data value and return it to the requesting task.

Camera Control

In the CAST Microsat, the primary mission is the CCD camera. In this spacecraft, the BBS will only be used for messages about stored images, and to store the images themselves. Weber State will write this application program.

IP and TCP

As an experiment, the TCP/IP suite of protocols will be ported to the Microsat CPU. This will allow processors such as FTP to be used in lieu of the BBS for file downloading.

FTP

File Transfer Protocol, part of the TCP/IP suite. It will have access to the various data files.

TELNET

The telnet protocol can be used in the TCP/IP suite to pass a stream of characters between a keyboard user and a program, and would thus serve as an alternate path to the BBS.

Implementation Schedule

The software group is aware of the tight schedule for the Spot 2 launch, and the need to balance the desire to do all of the above but be ready for testing in a few weeks. Therefore, we've separated tasks into several groups.

The highest priority group contains those things which must be present if anything useful is to be done with the spacecraft. This group includes the kernal, the HDLC driver, the AX.25 handler, the AART driver. To allow any reasonable development of the applications, these tasks must be present in their near-final form for testing and launch. This group also includes the bootstrap ROM code, which is being implemented by Hugh Pett in Canada, and is beyond the scope of this paper.

The next priority group are the applications which are required to run the spacecraft. Here, very simple temporary modules will do with the fancy ones coming later. The simple ones will be done ASAP; the fancier ones will be done before launch, time permitting. This group includes telemetry and housekeeping and memory wash. It also contains a very rudimentary BBS, with put-message, read-message, and list-message commands. The first BBS will also place its messages in linear memory, not in files.

Next are those things which are required before the final, fancy applications can be written. This is primarily the virtual ramdisk task. Finally comes the additional access methods such as TCP/IP.

Wrap up

This paper has discussed the current thinking in the area of Microsat operating software. Work to implement these plans is underway, some C code has already been run on the wire-wrap CPU prototype. Our choices of high level language, multi-tasking operating system, and implementation methodology have changed the project from impossible to merely difficult.

Acknowledgments

Thanks to Lyle Johnson WA7GXD for another fine hardware design, Chuck Green N0ADI for yeoman work at the wirewrap table; Skip Hansen WB6YMH for his work on the I/O drivers, Hugh Pett for the bootstrap loader; Martin Sweeting G3YJO and Jeff Ward G0/K8KA for past, present, and future collaboration on what a pacsat should do and how it ought to do it. Thanks to Jan King, W3GEY and Gordon Hardeman, KE3D, for stirring up this new project with the initial Microsat idea.