ON-BOARD SOFTWARE FOR THE MARS PATHFINDER MICROROVER

Jack C. Morrison
Tam T. Nguyen

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109

ABSTRACT

The Pathfinder Micro-rover Flight experiment will perform engineering and science experiments on the Martian surface to pave the way for future Mars missions. The rover is controlled by a 1970's era microprocessor. Its on-board software, while in some ways a typical embedded-system design, has to deal with some unusual constraints.

The rover will be operating in a harsh and mostly unknown environment, with limited electrical and processing power, accessible only via a limited-bandwidth communication link with long time delays. The software design is driven by these factors to provide reliability in the face of hardware, software, and operational failures, flexibility to allow adaptation and reconfiguration, simplicity, predictability, and visibility into its internal state and the external environment.

This paper describes the overall software structure, and details some of the more interesting features of the design, including error handling, power control logic, and navigation with hazard avoidance. The development environment is also described, including the use of world-wide-web-style hypertext to provide quick access to the collection of documents that accumulate in a software project.

MISSION TO MARS

The Mars Pathfinder Microrover, a 10-kilogram robotic vehicle, will perform engineering and science experiments on the Martian surface, and pave the way for future Mars exploration. Due to the communication time delays between Earth and Mars, and the unpredictability of the surface environment, the rover must operate semi-autonomously based on traverse waypoints and high-level commands from a human operator.

Research robots like the Pathfinder rover's predecessors often sport state-of-the-art processors, but a flight project requires proven, radiation-hardened components and imposes a severe power and weight budget. With the limited financial budget of a low-cost flight project on top of that, the result is a spacecraft with a difficult mission, a short development schedule, and a "computationally-challenged" central processing unit.

The rover's software architecture is motivated by several ideals intended to meet the mission goals while addressing all of these limitations:

- Reliability: handling failures of non-essential hardware components and unexpected environmental conditions. Time-critical anomalies (such as power drains and obstacle contact) must be handled without operator intervention. Software failures must be protected against occurring, and recovered from if they do occur.

- Flexibility: adapting to changes in the rover's hardware and environment. Modifications to behavior should be autonomous or easily commandable. Where possible, the rover is self-calibrating.

- Simplicity: in general, the simplest acceptable approach to each requirement or design problem is chosen. Besides being more reliable and flexible, simpler solutions are easier to test and have more predictable behavior. Not to mention taking less time to develop.

- Visibility: concisely but completely reporting the current hardware and software state, in particular unexpected conditions.

GENERAL SOFTWARE ARCHITECTURE

The rover is so constrained by electrical and processing power that it literally cannot "walk and talk" (run drive motors and communicate over its radio modem) at the same time. A multitasking executive would reduce performance without adding much capability. So instead, the software is organized as a single control loop, with interrupt handling for a few asynchronous "reflex" events (such as bumper contact) to which the rover must react quickly. This loop dispatches periodic functions (e.g. thermal management, automatic vehicle health checks, and command upload requests) as indicated by software timers, and invokes command handlers as directed by the uploaded command sequence.

Command handlers follow a common format:

- extract and validate any command parameters
- verify that the command is allowed based on the current rover state (e.g. mission phase and power availability)
- set a timeout limit for completion of the command
- perform the command
- format and send results telemetry
- return a completion status

NAVIGATION AND HAZARD AVOIDANCE

Perhaps the most unique component of the on-board software, for a spacecraft, is the waypoint navigation and autonomous hazard avoidance logic. Since the rover is normally directed along an operator-specified path based on 3D stereo images from the Pathfinder lander, it doesn't need to be a robust maze-solver - but it does need to watch for and deal with unexpected problems while moving.

The basic procedure for traversing to a waypoint is to drive forward in short segments, stopping for proximity scanning between each segment. In the absence of hazards, the rover arcs (at one of a small number of fixed-radius turns) toward the goal position. Inertial sensors and wheel encoders are used to dead reckon the vehicle's location.

While moving, other sensors (such as inclination and motor currents) are sampled for hazard detection and telemetry. Several conditions are monitored:

- reaching an operator-specified timeout limit
- sensor reading outside safe limits, from a table based on a risk-level parameter
- being too close to, and heading toward, the lander
- physical contact sensing

While stopped in between segments, various low-rate operations are performed when due, including a lander communication check "heartbeat" and transmission of buffered telemetry data. The rover then uses its optical proximity scanning system to look for obstacles (such as tall rocks or dropoffs). The scanner consists of infrared lasers and CCD imagers, which are used to build a sparse and approximate map of terrain height in front of the rover.

The map is searched for the following hazard conditions:

- missing data (inside the minimum required detection range; indicating a possible drop-off)
- height difference between any two adjacent detections above a (risk-level-dependent) threshold (indicating a rock or hole)
- height difference between lowest and highest detections above a higher threshold (indicating a steep slope)

If any proximity hazards are found in the map, the rover turns in place about its center as needed to avoid them. It first turns in whichever direction has the smaller required avoidance turn, or toward the goal if there's no preference. On any subsequent turns (after re-scanning) at the same location, it continues to turn in the same direction, to avoid bouncing back and forth between two hazards. Non-proximity hazards, such as contact and sensor readings outside safety limits, result in aborting the traverse - unless the operator has given the rover permission to deal with these itself. In that case, it backs up, turns, and tries again. Any time it has turned away from a hazard, the rover drives straight for a few segments before arcing back toward the goal.

Normally, the rover tries to keep far enough away from any hazards to allow it to turn around if it should get into trouble. The software has an option, however, to "squeeze" through narrower paths for a limited distance. If it runs into obstacles or fails to reach a clear region beyond this distance, it backs straight out and turns away from the narrow path looking for another alternative.

"Rock finding" is a modified driving mode, in which map data indicating a prominent rock triggers a behavior to center the rover heading between the edges of the rock using proximity sensing. If the destination waypoint is reached first, the rover performs a spiral search for a rock. The operator can then command the rover to turn 180 degrees and back its spectrometer onto the rock surface.

STATE INFORMATION

Nearly all of the global data is allocated in one of two structures - the volatile state area and the persistent state area. The volatile state data is initialized at each wakeup, and includes latest sensor readings and navigation control states.

The persistent state data is the rover's "long-term memory." It is loaded from EEPROM (electrically-eraseable programmable read-only memory) at wakeup, and the EEPROM image is updated regularly as the state changes. This data includes mission phase, vehicle location, odometry, and long-term time limits.

Persistent state data is the only part of the EEPROM that needs to be rewritten enough times to be in danger of exceeding the guaranteed lifetime of the memory during development and the mission. Enough space is therefore reserved for several copies of the state data, and the active copy is accessed indirectly through a pointer stored separately in EEPROM (this pointer is itself updated infrequently, so it can stay in a fixed location). A count of the number of rewrites is updated and stored with the persistent state; if this count reaches 95% of the guaranteed EEPROM lifetime (which is 1/10th the nominal lifetime), the state data is automatically relocated to the next allocated space. The pointer is updated accordingly, and the write count reset to one.

POWER CONTROL AND DEVICE MONITORING

Each controllable device, and each input sensor, is associated with a device status value indicating the health of that device. The value is automatically adjusted if anomalous behavior (such as an out-of-range input) is seen from that device, typically during periodic health checks, and adjusted back if normal operation is observed. With several steps between "normal" and "failed" states, the rover can autonomously disregard suspicious sensor data but recover from transient failures. At cold start, the status for failed devices is adjusted to that they have one chance to show that they are now working properly.

If a sensor reaches a "failed" state, the rover avoids using that sensor to influence its behavior. If possible, it uses an alternate source of information as a fallback. For example, it one temperature sensor fails, it may use a nearby temperature sensor instead. If a potentiometer indicating motor position fails, the rover can fall back to time-delay-based actuator control. However, actual sensor readings are always returned in telemetry - regardless of whether the rover trusts that data.

If an actuator devices reaches a "failed" state, the rover rejects operator commands that depend on that device functioning. The device status values are reported with health check telemetry, and can be modified directly by operator command. Special values allow the operator to force a device into a "good" or "failed" state which the software won't autonomously alter.

Whenever a device is turned on, the rover monitors the electrical current drawn by that device (after a short delay to allow for transients). If this current exceeds a device-specific limit, the rover sets the device's status to an additional special "failed" state that prevents that device from being turned back on. It can even mark a device with this "keep off" state if powering the device causes a system reset or trips the brown out-protection power monitor circuit. This logic reduces the chances of running down batteries or damaging electronics should a device short-circuit. The "keep off" state can be cleared (or set) by operator command.

Device status was a convenient capability during testing, because at any one time, some part of the vehicle was often missing. By setting the corresponding devices as "failed", the software could operate while ignoring invalid data from missing sensors.

CONTROL PARAMETERS

Numeric "constants" in the rover software are implemented in three forms. Those that are sure to remain fixed are coded as compile-time constants ("#define" preprocessor symbols or "enum" compiler constants).

"Constants" that will probably remain fixed, but could conceivably need to be altered to deal with unexpected conditions, are coded as "const" values, resulting in runtime constants stored in non-volatile memory with the code. Since the values exist at a single location that is identified in the linker map, these can be readily patched before or during the mission. A common use of patchable constants are the time delays needed between various control steps.

Finally, for a limited set of parameters that the operator might need to alter during the course of the mission to alter vehicle behavior, run-time variables or "control parameters" are provided. A dedicated operator command allows these parameters, which are part of the persistent state data maintained in EEPROM, to be easily changed. This "set parameter" command contains a table of valid limits for each control parameter to prevent acceptance of improper values. Some examples are navigation algorithm options, temperature thresholds for heating logic, and periodic processing rates. In addition to "behavior modification"-type parameters, some control parameters are used to store calibration data, such as zero offsets for position sensors, that can be updated late in the integration test phase based on experimental calibration measurements using the final flight hardware.

ERROR HANDLING

Two mechanisms handle error situations: an error reporting capability, and error state flags. The reporting function is invoked whenever a new anomaly is detected. An error ID unique to the source code location (8 bits indicating the module and 8 bits indicating an error call within that module), along with a severity level and a few words of optional associated information, are passed to the error reporter. If the severity level passes the current reporting threshold, this data is time-tagged and stored in a telemetry packet that is sent at the completion of the current command (or periodically, if no commands are executing). The error packet has a limited size; if it becomes full, additional error reports are counted but the associated data discarded. The severity threshold and overflow treatment prevent flooding of the telemetry channel with useless data in the event of a severe problem.

A utility program can quickly locate the program source corresponding to a specific error code, explaining the reason for and context of the error, and indicating the interpretation of the associated data. Pinpointing the error call directly in the source code eliminates the danger of an out-of-date or incomplete error code definition document, not to mention the work of maintaining one.

In addition to the error event reporting, eight error state flags track specific classes of error conditions, such as device failure, invalid command, and command execution timeout. They are set when corresponding problems are detected, and allow a limited form of conditional command execution, since most commands are skipped if error states are active. All flags, or a selected set, can be cleared by operator command. A group of interdependent commands within an upload sequence are typically sent starting with this "clear error" command so that they can proceed regardless of an irrelevant earlier anomaly; any failures seen during execution of the group causes the rest of that group - up to the next "clear error" command - to be skipped. The operator also has the option of masking out specific error state flags, to prevent a particular type of error from affecting command execution.

ROVER LITE

The rover computer has 160K bytes of non-volatile EEPROM memory, used to store the software, persistent state data, contingency command sequences (in the event of communication loss), and telemetry that can't be immediately sent to Earth via the Pathfinder lander (e.g. at night, when the lander may not be operating its modem). This EEPROM is particularly convenient during development, since software updates are easily downloaded, but remain resident across power downs. It will also allow patches to the rover software to be sent after launch.

However, there is some concern about the radiation tolerance and reliability of these devices. The first 16K bytes of the memory space is therefore populated by (expensive) radiation-hard non-erasable PROMs. This area contains the boot code first executed when the rover wakes up. (Being primarily solar powered, the rover normally shuts itself off every night.). The boot code computes and validates checksums on the EEPROM contents. If the EEPROM looks okay, control is passed to the normal EEPROM-resident software. Otherwise, a stripped-down rover control program contained entirely in the high-reliability PROM takes over.

This scaled-back program, known as "Rover Lite", can accomplish a subset of the mission objectives without depending on non-volatile storage. It includes the full communication protocol, some diagnostic and memory patching capability, and low-level device control. About half of the normal rover commands are supported in some form, along with a special "drive" command - in place of high-level navigation - allowing a sequence of vehicle motions to be programmed, something like a remote-controlled toy.

SOFTWARE TOOLS

The only software purchased explicitly for the rover project was a PC-hosted C compiler/assembler/linker/debugger package for the 8085 target architecture, and a PC emulator allowing that software to be run from UNIX workstations. The remaining development environment is based on common facilities - such as MAKE, RCS, AWK, and text editors - that are either freely available or part of the standard UNIX environment. Where appropriate, components of the target software itself were acquired from free Internet sources. These included the 8085 debug monitor, and CRC-16 and Reed/Solomon Error Detection and Correction algorithms.

ONLINE DOCUMENTATION

A new technique that has proven to be very useful is the collection of the assorted development documents that every project accumulates into web pages using a Hypertext Markup Language (HTML) index. These documents include such things as memory layouts, "how-to" notes for startup, shutdown, calibration, and other procedures, descriptions of utilities and development tools, coding standards, and delivery release notes. Both informal notes and official documents are accessible as web pages. Many are left as plain text files, but some have been formatted with HTML commands to allow instant access to sections of a large document or access to related documents through hypertext links. The HTML format also allows inclusion of images such as diagrams and drawings.

Having these documents available on-line in a standard format makes them readily available to the developers themselves as well as others on the project, whether they are using Unix workstations, Macs, or Windows PC's. The files can be kept up-to-date much more easily than paper documents, and there is no danger of someone working with an obsolete version of, say, an interface specification. They can be called up when needed during testing at any location that has Internet access, saving time and guesswork. It is at least hoped that this will save paper as well.

DATA GENERATORS

The rover software employs several related data tables and identifier lists to work with devices, sensors, and operator commands. In order to reduce the effort of changing these tables, and to eliminate the risk of tables being out of synchronization, scripts using the UNIX "AWK" program were built to generate the tables and identifiers from description files. For example, one file describes all of the analog sensors - multiplexor addresses, input gain settings, minimum and maximum expected values, etc. - and is processed into a list of sensor identifier codes, encoded tables used by the sensor input functions to setup and sample the sensor data, limit tables used by health check functions to validate sensor operation, and so on. Several times during development, there have been hardware changes that were quickly incorporated into the software by simple updates to the tables or the generator functions, without worrying about missing some corresponding change in a forgotten source file.

TEST UTILITIES

Development risk was reduced by implementing temporary software to stand in for external interfaces, eliminating dependence on other subsystems. Especially useful is the "simulator" for the lander spacecraft, which communicates with the rover by radio modem. Besides supporting tests of the modem communication protocol itself, this program accepts operator commands in text form (typed in or read from a command file) or in the official sequence file format (as generated by the Rover Control Workstation) and sends them to the rover. It also receives telemetry data from the rover, storing it on disk and optionally relaying it through a socket connection to other test software. This lander simulator can be run either in a window on a UNIX workstation, or on a PC (such as a laptop used for remote testing).

Another utility program can extract data from telemetry packets, and thus stands in for the ground data system, providing real-time (when connected to the lander simulator telemetry socket) and after-the-fact (when fed stored data files) viewing of telemetry. Related programs allow conversion of extracted image data, both normal and compressed, into a standard format for viewing and analysis.

These facilities have allowed end-to-end testing of the rover before the actual control workstation, lander, or ground data system were ready for integration. When such integration did take place, the rover team could was confident that its side of the interfaces had already been verified.

A simple downloadable menu-driven diagnostic program also proved to be worth far more than its development cost. This utility provides a thorough but easy-to-run regression test after hardware changes and environmental testing, a convenient facility for direct control of rover devices, and a "second opinion" on sensor or control anomalies observed by the operational rover software.

THE PATH AHEAD

The Pathfinder spacecraft is scheduled to launch in December 1996, arriving at Mars in July 1997. During integration testing so far, the microrover software has performed well, often demonstrating its flexibility and reliability when faced with changes in plans and hardware failures. Perhaps the biggest complaint is that it sometimes thinks *too* much, overriding our commands when we forget to tell it not to worry about some component that was temporarily removed. Fortunately, by the time the rover rolls out onto the surface of the red planet, we will be well-practiced in the art of cooperating with a semi-autonomous rover, and ready to explore together.

ACKNOWLEDGMENT

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.