Extending Command and Control for GOLF

Heimir Thor Sverrisson
W1ANT / TF3ANT

w1ant@arrl.net

Abstract

This work describes how existing AMSAT [2] software infrastructure has been adapted and extended to accommodate the Command and Control requirements of the new GOLF-Series Satellites.

The new GOLF [3] series satellites are three times bigger than the previous AMSAT Fox [5] satellites and are considerably more complex. One of the main differences is that GOLF has Attitude Detection and Control System ADCS [6], separate microwave communication equipment and a much more complex solar panel and power management system.

The existing Command and Control [1] software was not designed to deal with these new systems and therefore the work described here was taken on.

The main new feature is the ability to send Multi-Part Commands (MPC) to the spacecraft, which is necessary to handle the requirements of the new systems.

1 System overview

The overall system is shown in figure 1. It consists of a GOLF satellite, two existing software components and the new Multi-Part Command system (MPC). The existing command and control software is called AMCOM and will be described in more detail below. FoxTelem [4] is the existing software used to receive telemetry from the spacecraft, which has been extended to support a back-channel for status of the new Multi-Part Commands.

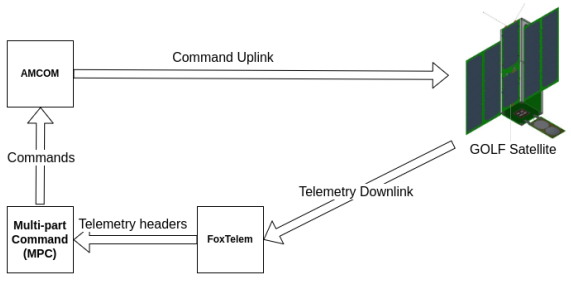

Figure 1. System Overview

2 Existing AMCOM Command Structure

AMCOM has several command types, but the one of interest here is the Software Command. This type is used to send commands that can carry only up to four parameters, each of which is a 16-bit integer.

The structure for Software Commands is shown in table 1.

Offset	Value	Remarks
0	Reset Number LSByte	6 byte header start
1	Reset Number MSByte	
2	Time Since Low Byte	
3	Time Since Mid Byte	
4	Time Since High Byte	
5	Spacecraft Address	6 byte header end
6	0	Payload start
7	Namespace	
8	Command Low Byte	
9	Command High Byte	
10	Value-1 Low Byte	
11	Value-1 High Byte	
12	Value-2 Low Byte	
13	Value-2 High Byte	
14	Value-3 Low Byte	
15	Value-3 High Byte	
16	Value-4 Low Byte	
17	Value-4 High Byte	Payload end
18 – 49	Signature	32 Bytes (512 bit)

Table 1. AMCOM Software Command structure

When the command is sent to the spacecraft it gets expanded to 100 bytes by the Forward Error Correction (FEC) encoding, taking about 1 second at 1200 baud.

The Namespace field is used to group commands into different categories. Only a handful of namespaces are currently in use and the same is true for the commands within each namespace.

The commands can carry up to four 16-bit integers as arguments, or a total of 8 bytes.

3 Multi-Part Commands

The limited payload size was not a problem for the FOX series satellites, but with the new systems on GOLF, it is necessary to send commands with larger payloads. In particular, the ADCS system needs a complete set of orbital parameters that must be set as a single command.

In order reduce system risk and reuse existing software as much as possible, it was decided to implement new command types called Multi-Part Commands (MPC). These command types would be able to send payloads of up to 128 bytes in up to 16 parts of 8 bytes each.

Offset	Value	Remarks
7	Namespace	MPC if 0x80 bit is set
8	Command Byte	
9	max, $current$ SeqNo	Sequence Byte

Table 2. MPC modifications to the Software Command

To accommodate the MPC extension, the existing AMCOM command structure was modified as shown in table 2.

The Flight Software in the spacecraft only needs to check if the Namespace has the high bit set, i. e. is $\geq 0x80$. If so, it knows that this is an MPC command and will process it accordingly, as described below.

The Command field has been reduced to one byte, instead of the previous two bytes per Namespace. This is not a problem as 256 different commands per Namespace is more than enough.

3.1 Sequence Numbers

The modifications to the command structure above allow for up to 16 AMCOM commands, called Parts, to be sent as a single MPC command. The byte at offset 9, called the $SequenceByte$, now contains two 4-bit fields, the maximum sequence number of the Parts in the MPC command, and the sequence number of this particular Part.

An example of an MPC command with 5 parts is shown in table 3.

Sequence Byte	Remarks
0x40	max part is 4, this is part 0
0x41	max part is 4, this is part 1
0x42	max part is 4, this is part 2
0x43	max part is 4, this is part 3
0x44	max part is 4, this is part 4

Table 3. Example of an MPC command with five parts

Think of the first nibble (4 bits) as the maximum value the second nibble can have. The second nibble is the sequence number of this particular part, both are zero based.

3.2 Command Acknowledgments

To confirm the reception of each part, the MPC system uses a five bit field in the telemetry header that was previously unused called $transmissionStatus$. The most significant bit is called $isAck$ and the four lower bits are called seq, described in more detail below.

As each telemetry packet is constructed by the Flight Software, it checks if there are any MPC commands that have been received since the last telemetry packet was sent. If so, it updates the $TransmissionStatus$ field in the telemetry header to indicate the status of the last received MPC command. As the telemetry packets are sent several seconds apart, it is important that the $transmissionStatus$ be updated just before the packet that shows the most recent status is sent.

The value in the $TransmissionStatus$ field will have its high bit set if there is an MPC command in progress. The lower four bits will then contain the sequence number of the last part correctly received. When all parts have been received correctly, the high bit will be cleared and the lower four bits will be set to zero.

In case of an error on the spacecraft, the high bit will be cleared and the lower four bits will contain an error code $(1-15)$. The error code will be transmitted from the spacecraft until the next MPC command is started.

An example of the $TransmissionStatus$ field and the meaning of its bits in telemetry headers is shown in table 4.

isAck	seq	Meaning
True	$0x01$	Second part is missing
True	$0x03$	Fourth part is missing
False	$0x00$	Command executed

Table 4. Example of the Transmission Status

3.3 Ground Station algorithms

The procedure for sending MPC commands is shown in algorithm 1.

The Ground Station software has two separate threads of execution. These two threads share three variables: $isAck$, seq and $gotAck$. The first one indicates if there is an MPC command in progress on the spacecraft, the second one is the sequence number of the next part to be sent (in case retransmission is needed), and the third indicates that we have received an acknowledgment since we started on this MPC command. These common variables are reset with the procedure on line 5 which is called every time we start sending a new MPC command on line 12.

When $isAck$ is false, the seq variable will contain the result of the last MPC command, which is zero if the command was successfully executed or an error code $(1 - 15)$ if an error occurred.

The entry point is the SENDMPC procedure on line 11, which takes as input a list of parts that make up the MPC command. After resetting the shared variables, we wait for the next acknowledgment from the spacecraft on line 13.

The guard on line 14 checks if there is an MPC command in progress. If not, we send all parts of the MPC command to the spacecraft starting on line 15. If there is an MPC command in progress, we report an error on line 19 and the operator should try again later.

Note that we send all parts of the MPC command without waiting for acknowledgement. This is done to speed up the process as telemetry packets are only sent every few seconds and will not reflect the true state of the MPC command in the spacecraft until after all parts have been sent.

Also note that we can still receive an acknowledgment from the spacecraft during the transmission of a long MPC command, that will then set the $gotAck$ state variable to $true$.

Next we enter a loop on line 22 where we wait for the next telemetry packet from the spacecraft. On line 24 we check if we have received a packet with $isAck$ set to true. We only need to resend if we receive a sequence number is not equal to the number of parts sent modulo 16, because that indicates all packets have been received correctly. Note that if, for example, we are sending 16 parts, the sequence number when all parts have been received correctly will be $0xf + 1 \bmod 16 = 0$.

In case $isAck$ is $false$ we check if any acknowledgment has been received since we started transmission on line 30. If not we declare an error.

Next check on line 32 is for error code from the spacecraft and if that is not the case we are done sending.

The WAITFORACK procedure in algorithm 2 is running in the second thread of the Ground Station software. It waits until a telemetry packet is received and extracts the five bit $TransmissionStatus$ field from the header. It then sets the shared boolean, $isAck$, indicating if there is an MPC command in progress (based on the high bit of the field), and the sequence number of the first missing part (based on the lower four bits). As mentioned above, when $isAck = false$ the four lower bits will be zero if the MPC command was successful, or an error code $(1-15)$ if an error occurred.

The check on line 4 will allow us to capture any packet with the $isAck$ flag $true$ that is received during the sending period. This logic makes the detection of MPC activity in the spacecraft more robust because we insist on getting at least one such packet during the MPC command transmission. Otherwise, we might only get $0x00$ as $TransmissionStatus$ before we start sending and the same value after we've sent all parts, which would be exactly the same response as if no part of the MPC command was received.

It is implied that WAITFORACK will block until a telemetry packet is received, but will timeout after a reasonable time if no packet is received, returning an error condition that will terminate the send function.

In this code we assume that the DOWNLINKRECEIVEEVENT is used to synchronize the two threads. The $Wait$ called on lines 13 and 23 in algorithm 1 will block until the event is Set by the other thread. This is done when a telemetry packet is received on line 4 in algorithm 2. The $Clear$ method will clear the event so that the next call to $Wait$ will block.

We also assume that the shared variables, $isAck$, seq and $gotAck$, are protected by a mutex or similar mechanism for safe access from the two threads.

3.4 Flight Software algorithm

The Flight Software algorithm for receiving MPC commands is listed as algorithm 3. The main entry point is the RECEIVEPART procedure on line 25, which takes as input a single part of an MPC command. In line 26 we check if there is an MPC command in progress. If not, we initialize the state for a new MPC command on lines 27 to 30.

If there is an MPC command in progress, we check on line 31 if the $Namespace$ and $Command$ fields match the current command. If not, we reset the internal state on line 32 and report an error on line 33. The call to ERROR sets the value of $error$ to 1 and returns without storing this part. The error value is reported through the telemetry interface as described below.

If all is good, we store the part in line 36 by calling the STOREPART procedure. The STOREPART procedure on line 13 copies the payload of the part to a buffer on line 14. It then updates the bitmap of received parts on line 15 and computes the sequence number of the next part to be received on line 16 by calling the FIRSTGAP function.

Algorithm 1 Send MPC Parts

```
1:  isAck ← false
2:  seq ← 0
3:  gotAck ← false
4:
5:  procedure RESET
6:      isAck ← false
7:      seq ← 0
8:      gotAck ← false
9:  end procedure
10:
11: procedure SENDMPC(Cmds)
12:     RESET
13:     DOWNLINKRECEIVEEVENT(Wait)
14:     if ¬isAck then
15:         for all cmd ∈ Cmds do
16:             AMCOMSEND(cmd)
17:         end for
18:     else
19:         ERROR("MPC command in progress")
20:     end if
21:     isDone ← false
22:     while ¬isDone do
23:         DOWNLINKRECEIVEEVENT(Wait)
24:         if isAck then
25:             if seq = len(Cmds) mod 16 then
26:                 isDone ← true
27:             else
28:                 AMCOMSEND(Cmds[seq])
29:             end if
30:         else if ¬gotAck then
31:             ERROR("No response from spacecraft")
32:         else if seq ≠ 0 then
33:             ERROR(seq, "MPC Command failed")
34:         else
35:             isDone ← true
36:         end if
37:     end while
38: end procedure
```

Algorithm 2 Downlink MPC Acknowledgments

```
1: procedure WAITFORACK
2:     DOWNLINKRECEIVEEVENT(Clear)
3:     isAck, seq ← WAITFORTRANSMISSIONSTATUS
4:     DOWNLINKRECEIVEEVENT(Set)
5:     if isAck then
6:         gotAck ← true
7:     end if
8: end procedure
```

On line 17 we check if all parts have been received. If so, we can execute the command by passing the complete buffer to an external EXECUTECOMMAND function on line 18. In order to guarantee that at least one telemetry packet is sent with the $isAck$ bit set, we synchronize with the telemetry thread. By clearing the ACKSENTEVENT on line 19 we ensure that we block on line 20 until a telemetry packet with the $isAck$ bit has been sent.

We then reset the internal state on line 21 by calling the RESETMPC procedure. The telemetry interface described below will then indicate that the MPC command was successful.

The FIRSTGAP function on line 5 computes the sequence number of the first missing part by checking the bitmap of received parts. Note that bitmap is a 32-bit integer, to accommodate a command with up to 16 parts.

This method allows for parts to be received out of order and also allows for retransmission of missing parts. The missing parts are requested from the lowest ordered to the highest ordered as seen in the code in FIRSTGAP.

3.5 Flight Software Telemetry Interface

The Flight Software interface to telemetry is shown in algorithm 4. Every time a telemetry packet is constructed the GETTRANSMISSIONSTATUS function is called to get the bitmap value to put in the $TransmissionStatus$ field of the telemetry header.

If there is an MPC command in progress, the function returns $0x10$ OR-ed with the sequence number $(0-15)$ of the first missing part of the command, indicating the first gap in the received parts. If an error has occurred, the function returns the error code $(1-15)$. If there is no MPC command in progress and no error, the function returns $0x00$.

Note that the error code will persist until the next MPC command is started.

On line 4 we synchronize with the MPC receive thread that will be blocked in line 19 of algorithm 3 until at least one telemetry packet has been sent to the Ground Station with the $isAck$ bit set.

Also note that the telemetry interface is being called by a different thread from the one that is receiving the MPC commands. Therefore, the shared variables, $inProgress$, $nextAck$, and $error$ must be protected by a mutex or similar mechanism.

3.6 Ambiguous acknowledgment

There is a problem with the simple algorithms described above. In the case of an MPC command with exactly 16 parts, a missing Part 0 will generate the same acknowledgment value as if all parts have been received correctly, because $16 \bmod 16 = 0$.

Algorithm 3 Receive MPC command

```
1: inProgress ← false
2: nextAck ← 0
3: error ← 0
4:
5: function FIRSTGAP
6:     firstGap ← 0
7:     while pBitmap & (1 ≪ (firstGap)) ≠ 0 do
8:         firstGap ← firstGap + 1
9:     end while
10:    return firstGap mod 16
11: end function
12:
13: procedure STOREPART(cmd)
14:    COPYTOBUFFER(partBuffer, cmd)
15:    pBitmap ← pBitmap | (1 ≪ cmd.seq)
16:    nextAck ← FIRSTGAP
17:    if nextAck = (cmd.maxSeq + 1) mod 16 then
18:        EXECUTECOMMAND(nmSpace, command,
   partBuffer)
19:        ACKSENTEVENT(Clear)
20:        ACKSENTEVENT(Wait)
21:        RESETMPC
22:    end if
23: end procedure
24:
25: procedure RECEIVEPART(cmd)
26:    if ¬inProgress then
27:        RESETMPC
28:        inProgress ← true
29:        nmSpace ← cmd.nmSpace
30:        command ← cmd.command
31:    else if cmd.nmSpace ≠ nmSpace ∨
   cmd.command ≠ command then
32:        RESETMPC
33:        ERROR(1,"command mismatch")
34:        return
35:    end if
36:    STOREPART(cmd)
37: end procedure
38:
39: procedure RESETMPC
40:    inProgress ← false
41:    error ← 0
42:    nextAck ← 0
43:    ACKSENTEVENT(Clear)
44:    pBitmap ← 0x00000000
45:    nmSpace ← 0
46:    command ← 0
47: end procedure
```

Algorithm 4 MPC Telemetry Interface

```
1: function GETTRANSMISSIONSTATUS
2:     if inProgress then
3:         transmissionStatus ← 0x10 | nextAck
4:         ACKSENTEVENT(Set)
5:     else if error ≠ 0 then
6:         transmissionStatus ← error
7:     else
8:         transmissionStatus ← 0x00
9:     end if
10:    return transmissionStatus
11: end function
```

This condition can easily be address in the Ground Station software because the missing Part 0 acknowledgment will be received before all parts had been sent, which can never happen for the acknowledgment of the final part.

This is a consequence of needing to communicate 17 states while only being able to set 16 values, since we have a four bit field. Other acknowledgment schemes have similar problems, i.e. if we were to send the sequence number of the last consecutive part received $(0 - 15)$ instead of the next to be sent, we would have no way of indicating that we need to resend Part 0.

4 Changes to existing Software

To implement the MPC extension method described above, several changes will have to be made to existing software. These changes are discussed below.

4.1 AMCOM

AMCOM itself will need very minimal changes, as the structure of these new Multi-Part Commands is the same as the current Software Commands. Code that is very similar to what is already implemented for external CAN commands will be added to AMCOM. When used with MPC, AMCOM is practically acting in a worker-mode. AMCOM will still be used for actual transmission of the commands to the spacecraft in addition to Authentication, Authorization, Forward Error Correction and so on. This approach minimizes risk by reusing existing tried-and-true software.

4.2 Flight Software

The Flight Software in the spacecraft must be extended to process commands with $Namespace \geq 0x80$ by setting aside storage for 8 bytes per part of a Multi-Part command to store the payload while parts are being received. This is a maximum of 128 bytes for a command with 16 parts.

The $TransmissionStatus$ logic described above must also be implemented by the existing Telemetry module, for inclusion in all the telemetry packet headers sent to the ground.

Finally when all the Parts of a Multi-Part command have been received, the Flight Software must take the action needed, i.e. update the orbital parameters of the ADCS.

4.3 FoxTelem

FoxTelem must be able to communicate the *TransmissionStatus* field to the MPC Sending Software. This change has already been implemented by sending the whole telemetry header to an external program, like the MPC, over a TCP/IP socket.

4.4 AUTHGEN

The AUTHGEN program is now used to create authorization for users to use certain commands in AMCOM for certain spacecraft. The current scheme will be extended to authorize individual Multi-Part Commands as described below. Because the construction of the actual payload of the Multi-Part Commands is more complicated than the static nature of the existing commands, the payload data will be managed further by the MPC Ground Software.

5 Command interactions

In order to safely operate the spacecraft with this extension of the command structure several constraints will have to be considered in the Flight Software.

5.1 Concurrent commands

We need to consider the case of more than one Ground Station sending commands at the same time. If they do not overlap in time they can arrive correctly at the spacecraft.

As the Namespace value in the commands can be used to select the code path in the Flight Software there should not be a problem if a regular command arrives between parts of a Multi-Part Command. This simply assumes that the state storage for Multi-Part Commands is not affected by the processing of the regular commands and they can be executed independently.

A different Multi-Part Command received while another one has not been fully processed should cancel the first one and clear all partial states for that command as implemented in the algorithms listed above. This would also happen in the valid scenario of a Ground Station operator changing their mind in the middle of a transmission and deciding to send a different command. Note that an AMCOM command must be signed correctly to be accepted by the Flight Software.

5.2 Timing of commands

Using synchronization of timing between the Ground Station and a spacecraft available from FoxTelem, a rather reliable monotonically increasing timestamps can be sent by AMCOM via MPC. This will allow the Flight Software to compare timestamps of a part to the previously received part. If too much time has elapsed from the previous one all previous parts should be discarded as described above for partial command cancellation and this part should be regarded as a start of a new Multi-Part transmission (including the command value itself).

Partial Multi-Part Commands do not need to be kept in memory indefinitely. If command transmission of a Multi-Part Command has not been completed before LOS, it should be discarded and re-transmitted as a whole later. There is no need to check for this condition periodically, the mechanism described above for timeout will take care of it as soon as the next part is received (maybe not until the next pass).

6 Configuration files

6.1 AUTHGEN

In order to fit the Multi-Part Commands into the authorization structure of AMCOM a new command type, `mcmd`, will be added to the `SatCmds` file. One item is added per Multi-Part Command in the same way as Software Commands, allowing for the same authorization granularity as other commands.

The `SatCmds` currently has multiple sections by type of command. These sections are each in a CSV format, but with different fields per command type. It needs to be decided if a new more flexible format, i.e. JSON, should be used instead for this configuration.

Regardless of the format, new data types will need to be introduced to handle the data payload of the Multi-Part Commands. Table 5 shows the suggested data types.

6.2 MPC

The payload data for the Parts sent from MPC to AMCOM will be read from a CSV file with the header as specified in the `SatCmds` file. This also allows for these payload CSV files to be generated by other programs, i.e. converters from TLE, etc.

Data Type	Description
`i8`	8-bit unsigned integer
`i16`	16-bit unsigned integer little endian
`i32`	32-bit unsigned integer little endian
`i64`	64-bit unsigned integer little endian
`f32`	32-bit IEEE floating point
`f64`	64-bit IEEE floating point
`str`	utf-8 string up to 8 bytes

Table 5. Suggested data types for MPC payload

6.3 Header file generation

With the added complexity of Multi-Part Commands it is important to automatically generate the header files used by the Flight Software to define the command values and the data structures used to access the different fields in the payload. The input to the header file generator are the configuration files discussed above.

7 Conclusion

The new Multi-Part Command system allows for sending commands with up to 128 bytes of payload, which is sufficient for the new GOLF satellites.

The existing AMCOM software will be modified to support the new command types, while using the tried-and-true communication mechanisms and protocols.

The headers of the telemetry packets sent by the spacecraft will be modified to include the status of the Multi-Part Commands, thereby providing a back-channel for command acknowledgment and errors using the FoxTelem software.

The Flight Software in the spacecraft will also be modified to support the new command types, while keeping the existing functionality mostly intact, thereby minimizing the risk of reducing overall system stability.

References

[1] Burns Fisher, W2BFJ. A Modern Approach To Secure Commanding of an Amateur Satellite. In *Proceedings of the AMSAT 33rd Space Symposium, pages 37–42, 2015*

[2] AMSAT. Radio Amateur Satellite Corporation. `https://www.amsat.org/about-amsat/`. Accessed: 2025-09-20.

[3] Bob Davis, KF4KSS. GOLF-TEE Mechanical Design. In *Proceedings of the AMSAT 36th Space Symposium, pages 11–18, 2018*

[4] Chris Thompson, G0KLA/AC2CZA. Designing the Fox-1E PSK Modulator and FoxTelem demodulator. In *Proceedings of the AMSAT 36th Space Symposium, pages 27–41, 2018*

[5] Tony Monteiro, AA2TX. Fox Satellite Program Overview. In *Proceedings of the AMSAT 31st Space Symposium, pages 41–17, 2013*

[6] P.J. Haufe, et. al. Design and Verification of the Attitude Determination and Control Algorithms for the Source Cubesat. In *Deutscher Luft- und Raumfahrtkongress, 2023, DocumentID: 610141*. `https://www.dglr.de/publikationen/2023/610141.pdf` Accessed: 20225-09-20