GENERIC DECOMMUTATION CAPABILITIES IN THE SPACE FLIGHT OPERATIONS CENTER

Robin A. O'Brien

Member of the Technical Staff
Telemetry Software Group
Control Center Data Systems Development Section
Jet Propulsion Laboratory/California Institute of Technology
4800 Oak Grove Dr.
Pasadena, California 91109

ABSTRACT

A generic decommutation capability has been created as part of the Space Flight Operation Center's goal of developing a multi-mission telemetry system. Generic decommutation involves separating the algorithmic description for extracting data from the actual implementation of decommutation. This was done by creating a Decommutation Map Language, which allows mission designers to describe decommutation algorithms without the restrictions imposed by a standard programming language. A Decommutation Map Compiler converts this description into C code, which is then linked with a decommutation library to provide an executable decommutation program. So far, this approach has been used successfully to decommutate several different types of data.

INTRODUCTION

As part of SFOC's multi-mission goal, a decommutation system must require a minimum amount of work to be adapted for a new project, but it must not impose any serious restrictions on a Space Flight mission's data format. Several prototype efforts investigated decommutation implementations with this goal in mind; all worked with the premise that in order to make decommutation truly multi-mission, the data description and algorithm for decommutation must be separated from the actual decommutation implementation. The final product is based on a prototype initiated by William Luebkert at the Jet Propulsion Laboratory.

SFOC decommutation involves creating a decommutation "map" which describes the data and how it should be decommutated. This map, written in the Decommutation Map Data Language (DMDL) is converted into a format capable of interacting with decommutation

library routines to provide an executable decommutation program. For different data types or different missions, only the map needs to be changed.

DECOMMUTATION MAP DATA LANGUAGE

There are three primary functions within DMDL: describing a channel for decommutation, skipping data that is not being decommutated and **switching** on values in the data.

The major function of decommutation is to specify **channels**. A channel refers to a specific portion of data. It has an identifier (usually an alphabetic character followed by a string of digits, i.e. A-1010), a width (the number of bits in a channel), and an offset within the data. A channel is indicated in the data simply by stating a channel identifier. Decommutation begins at the first bit in it's input data (the 0 bit). This is the initial current bit position. When a channel identifier is encountered in a map that channel is decommutated and the current bit position moves to the first bit after the channel. By default, channel width is eight bits. The default width can be changed in a map, or it can be changed for an individual channel.

Of course, not all data in a given data block may be pertinent for decommutation. DMDL provides a way to skip data. A **SKIP** statement allows the user to move the current bit position forward by a specified number of bits. There is also a **DEFINE OFFSET** statement that allows the user to specify that the current bit position be moved to a specific bit offset in the data.

Decommutation in SFOC is rarely so straightforward that it can be described as a simple series of channels. To accommodate the more complex algorithms within SFOC, **switching** on data plays an important role. A switch statement in DMDL is very similar to a switch statement in many programming languages, and fits in readily with the needs of decommutation. A switch in decommutation means that at a specific location, different channels may exist depending on the value in the **switch variable**. A switch statement consists of **cases** which are possible values of the switch variable. DMDL provides a variable statement to create switch variables. A switch variable is an actual value extracted from the input data at some time before the switch statement is encountered. Data is extracted as it is for a channel, but the current bit position is not changed. A variable has a width like a channel, but may not exceed 32 bits. Switch statements can be nested.

In telemetry data, certain fields may be repeated within the input data. For instance, there may be 100 possible records for engineering data, but some channels may repeat after 10 records. To reduce redundancy, modulo operators are allowed in switch statements. In a case where there are 100 possible records, but only 5 different channel patterns, the decommutation map can use a construct like this

```
SWITCH variable % 5

        case 0 ...
        case 1 ...
        case 2 ...
        case 3 ...
        case 4 ...
ENDSWITCH
```

Obviously, this is much easier than listing a hundred different cases.

Finally, DMDL allows the user to organize the information into a sequence of blocks, similar to subroutines in a programming language. There must be a block named **MAIN**, which is the first block executed. From this block, the user is free to call other blocks as needed. Blocks can also be called from different case statements within switch statements.

Use of a decommutation map provides several advantages. First, it is a way to eliminate overhead. The decommutation library contains all modules needed to create an executable program, including the front end. A programmer only needs to maintain this library of C code routines. Second, it is much simpler to change channels in a decom map than in a C program. In the past, the sheer number of channels in data have led to errors in channel names, widths and location. These errors can go undetected until the actual channel comes up for scrutiny. With DMDL, the channels are simple to specify, simpler to spot check for errors, and DMDL's relative simplicity allows for the development of tools to verify channels. Finally, no extensive programming knowledge is required to create and/or maintain a decommutation map. While DMDL has constructs similar to those found in programming languages, it is geared more toward describing telemetry than in providing programming tools. The language has been kept as small as possible so it can be learned quickly by inexperienced users.

Appendix A contains a BNF grammar of the Decommutation Map Language. Appendix B contains a sample of an actual Decommutation Map.

HOW AN EXECUTABLE MAP IS MADE

Creating an executable decommutation program requires only two simple steps. First, the DMDL file is compiled using a **decommutation compiler**. Using UNIX's YACC (Yet Another Compiler Compiler) utility, the decommutation compiler converts DMDL to C code that uses the Decommutation Library routines. Syntactical errors in the use of DMDL are detected automatically by the YACC generated code. Semantic restrictions imposed on

the user (such as the maximum channel or variable width) are also detected at compile time. All errors are fatal errors, and processing halts when the first error is detected.

If the map compiles correctly, the output is a file of C code. A Decommutation library exists that contains all the necessary routines for decommutation including a **MAIN** procedure. The decommutation map blocks are converted to C subroutines and the prefix **map_** is added to each block name. Map semantics requires there be a block labeled **MAIN** in each decommutation map, and **map_MAIN** is called by the decommutation library routines to execute the map. This arrangement makes it relatively simple to add front and back end processing to the map, since the map processing is really being driven by the routines in the library. Figure 1 illustrates the steps taken to produce the executable program.

DECOMMUTATION IN MAGELLAN

The Magellan Project, formerly called the Venus Radar Mapper, is one of the first Space Flight Projects to use the SFOC implementation. For Magellan, a more complex method of creating and using maps was employed.

Map installation takes place in two parts. First, the map must be verified. The map is compiled with the decommutation compiler, although no output is produced. If the map can be compiled without errors, then the user can input various parameters concerning the maps validity, i.e. what type of data the map is good for, what time period the map is good for, etc. The map and its parameters are stored in a single file. Only these verified files can be used in the second step of installation, which creates a **decom daemon**. A daemon in the UNIX operating system is an asynchronous process that runs in the background. A decom daemon in SFOC is an asynchronous decommutation process. To create the daemon, the second installation procedure takes a verified file and compiles it with the decommutation compiler and then with the CC compiler and decommutation library to produce an executable file. The names of the daemons and their map parameters are entered as records in a Version Control Table file (VCT).

When decommutation is initialized, a decommutation parent process uses the records in the VCT to determine which decom daemons to start. For each data type being decommutated, a record from the VCT is selected and the daemon named in that record is executed and a queue is opened between the parent and the daemon. The parent will then route data of the appropriate type to the selected decorn daemons. Figure 2 illustrates the current decom daemon implementation.

When a daemon receives data, it verifies that the data fits the parameters specified in the VCT for this particular daemon. If the data does not fit the parameters, it will search the

VCT for a record that does. When the daemon locates such a record, it starts the daemon named in that record and passes the data along to it. The original daemon terminates itself after it sends the new daemon data. The decom parent process will now route data to the new daemon. Figure 3 represents how a daemon process is started.

ACKNOWLEDGEMENT

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

APPENDIX A
DECOMMUTATION MAP DATA LANGUAGE GRAMMAR

```
decom_map     -->    body | w_s body

body          -->    block | body block

block         -->    block_header list_of_stmts end_block
                     | block_header list_of_stmts end_block w_s

block_header  -->    BLOCK NAME '\n'

end_block     -->    ENDBLOCK

list_of_stmts -->    stmt | list_of_stmts stmt

stmt          -->    switch_stmt
                     | define_stmt
                     | block_call_stmt
                     | channel_stmt
                     | skip_stmt
                     | variable_stmt
                     | gen_chan_stmt
                     | '\n'

switch_stmt   -->    switch_spec w_s case_stmt end_switch '\n'

switch_spec   -->    SWITCH  NAME
                     | SWITCH NAME '%' NUMBER

end_switch    -->    ENDSWITCH

case_stmt     -->    single_case_stmt | case_stmt single_case_stmt

single_case_stmt-->  case_value_stmt list_of_stmts end_case w_s

end_case      -->    ENDCASE

case_value_stmt-->   CASE NUMBER '\n'
                     | case_value_stmt CASE NUMBER '\n'

define_stmt   -->    offset_stmt | width_stmt

offset_stmt   -->    DEFINE OFFSET def_op '=' offset_value

width_stmt    -->    DEFINE WIDTH  def_op '=' expr

def_op        -->    '+' | '-'

offset_value  -->    expr | header_value

header_value  -->    PRIMARY_LABEL
                     | PRIMARY_HDR
                     | SECONDARY_HDR
                     | TERTIARY_HDR
                     | QUATERNARY_HDR
                     | DATA
```

(*Note: The header values are specific offsets in the data that are particular to SFOC data)

```
block_call_stmt-->   CALL NAME

channel_stmt  -->    NAME '-' NUMBER
                     | NAME '-' NUMBER '(' NUMBER ')'
                     | NAME '-' NUMBER '(' ':' NUMBER')'
                     | NAME '-' NUMBER '(' NUMBER ':' NUMBER ')'

variable_stmt -->    NAME '(' NUMBER ')'
```

```
                        | NAME '(' NUMBER ':' NUMBER ')'

skip_stmt       -->     SKIP expr

gen_chan_stmt-->        GENERATE '(' NAME '-' NUMBER ',' NUMBER ',' expr ')'

expr            -->     NUMBER
                        | '(' expr ')'
                        | expr '+' expr
                        | expr '-' expr
                        | expr '*' expr
                        | expr '/' expr

w_s             -->     w_s '\n' | '\n'
```

(*Note: NAME and NUMBER are defined by the lexical analysis routines in the compiler)

APPENDIX B

DECOMMUTATION MAP SAMPLE

NOTE: Text on a line following a "#" are comments

```
##############################################
# main definition block
BLOCK MAIN
  DEFINE WIDTH = 8

  DEFINE OFFSET = SECONDARY_HDR

  skip 2*8                # jjw4
  H-0002          # scft_id
  H-0001          # data_source
  skip 9
  H-0023(1)               # FSS Checksum jjw5
  H-0022(2)               # ECS error jjw5
  skip 4
  H-0020(16)      # ert, days since epoch
  H-0021(32)      # ert, fractional days in ms
  skip 22*8

 RMFID(8)         # radar frame id
  skip 8
 RMF_REF(1)       # 0 = established   1 = not established
  skip 2
 TRICKLE_REF(1)   # 0 = established   1 = not established
  skip 6
 TRICKLE_INDEX(8) # use as switch variable to decom trickle
  skip 16
  DEFINE OFFSET = DATA
  DEFINE OFFSET += 272
  CALL AACS
#
  DEFINE OFFSET = DATA
  DEFINE OFFSET += 400
  SWITCH RMF_REF % 2
    CASE 0
      skip 8          # Skip minor frame ID  #dsh
      CALL RADAR
    ENDCASE
    CASE 1
      skip 120
    ENDCASE
  ENDSWITCH
#
ENDBLOCK
```

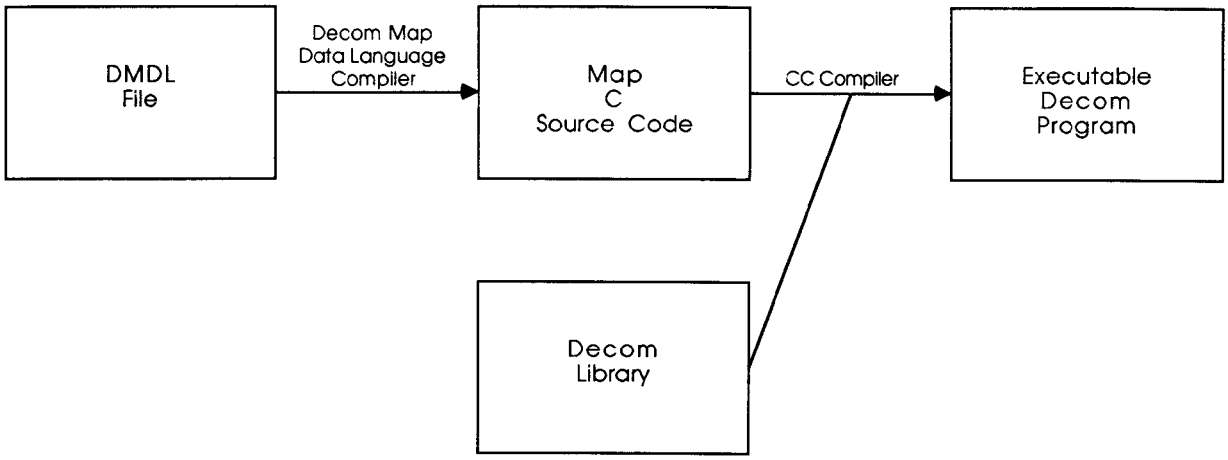

Figure 1 - Creating a Decom Program

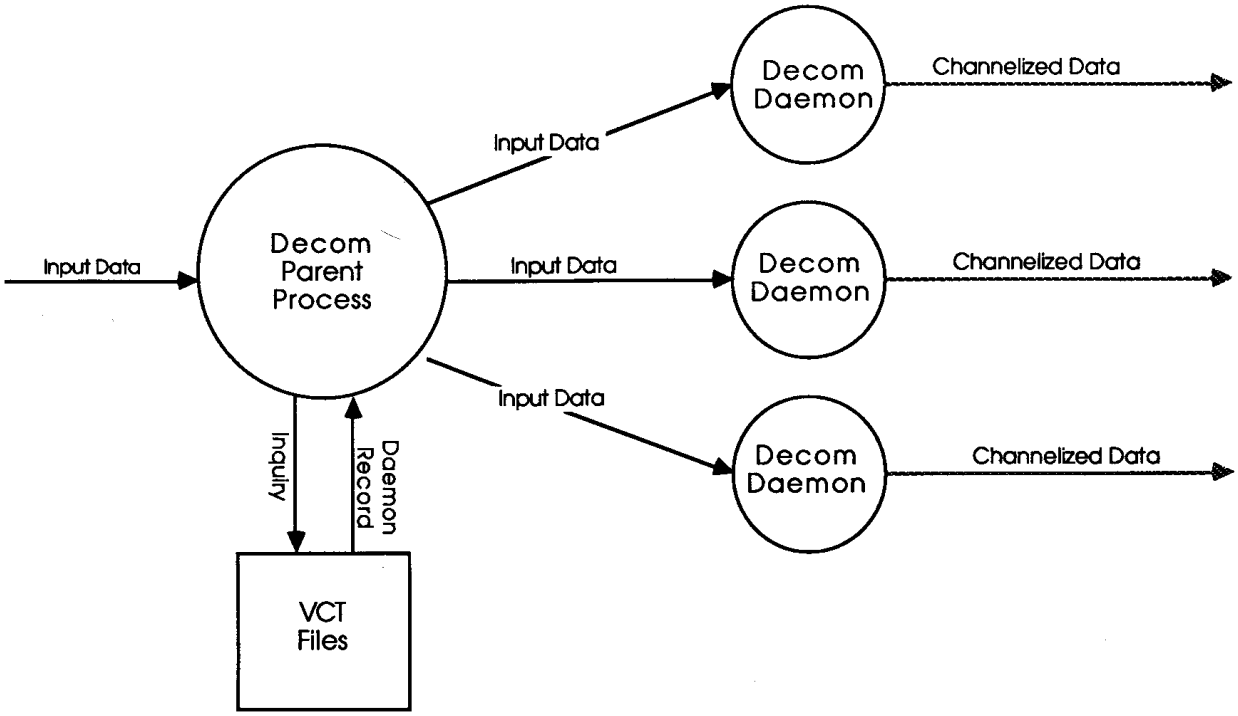

Figure 2 - Magellan Decommutation Implementation

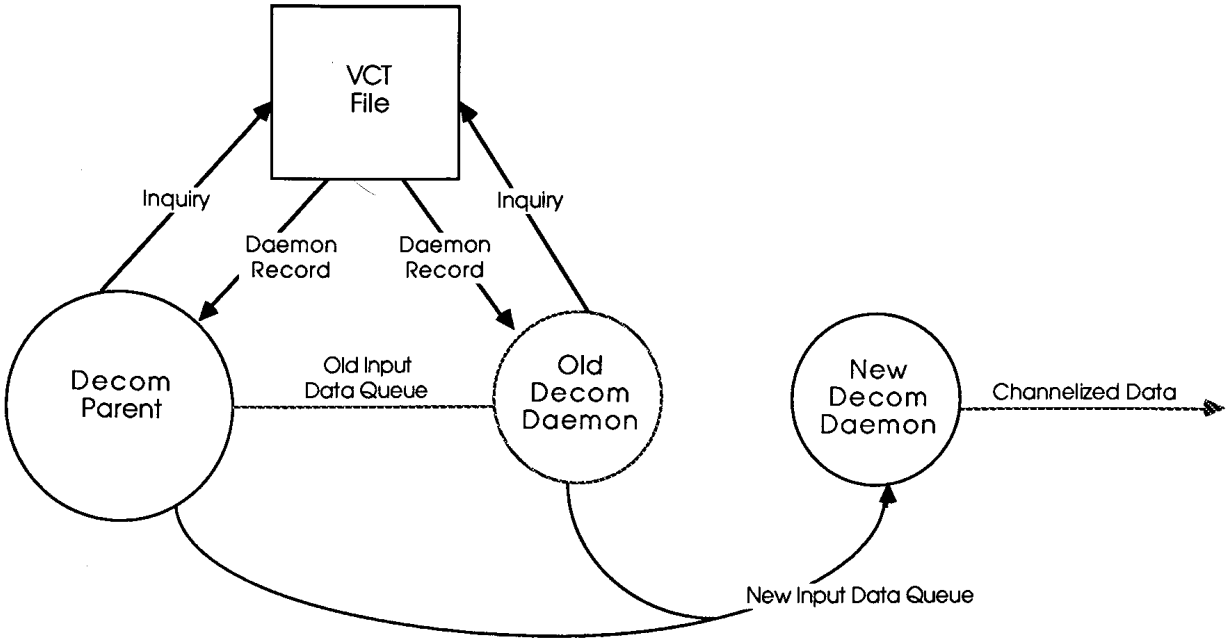

Figure 3 - Changing Decom Daemons