

ROD TEST STAND SOFTWARE

V0.0

Lukas Tomasek, 15 March 2000

1 Revision history

V0.0

- 6 March 2000- First not very complete draft (proposal).
- 9 March 2000 - fixed some typos (Fig.2 and 3).
- 15 March 2000 - added Fig. 1B.

2 Introduction

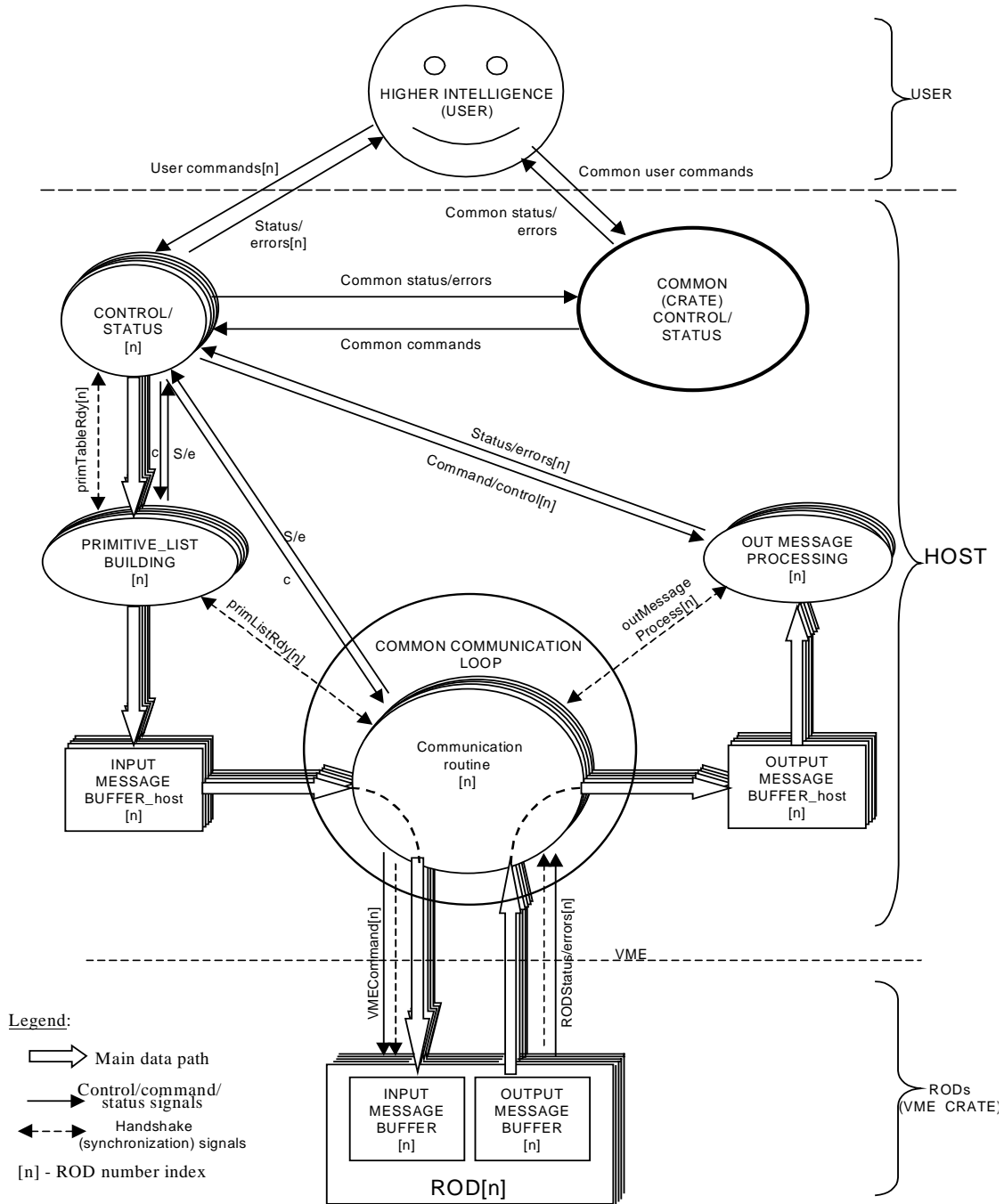
This document is mainly intended to describe ROD test stand (host) software as an amendment to the document "Communication Protocol and Primitive Execution", <http://www-wisconsin.cern.ch/~atsiod/shared.html> , where are explained the details of the communication between Host and MasterDSP and all main definitions.

Test stand hardware and software tools:

- VME crate, NI VME-MXI-2 and PCI-MXI-2 interface,
- RODs??
- PC with Windows NT operating system,
- LabWindows software

3 PROGRAM STRUCTURE

3.1 High level layout - process diagram



Note: InfoBuffer, DebugBuffer and ErrorBuffer readout not shown.

Fig. 1A High level layout of the test stand software - process diagram

All threads except **CommonControl/Status** and **CommonCommunicationLoop** are specific for each ROD[n] in the crate. This is emphasized by index [n], where **n** means ROD number in the crate.

...

3.2 CommonControl/Status thread

This is the main program thread, so its base task is to create all the other "subthreads" after program startup.

Since this thread is "practically" the only one shared by all RODs in the crate, it can also periodically collect and display (in the "Common status/error window") important "crate" information (for example logical OR of all errors in the system etc.).

The User can issue by this thread common commands like initialization or software reset for all RODs.

...

3.3 Control/Status[n] thread

The main target for this thread is:

- to interact with the user through LabWindows GUI. It means providing all important or requested status/error/data information ("ROD[n] status/error window") about "inferior" ROD[n] and the threads attached; and "user friendly" environment for creating/editing PrimitiveLists ("PrimitiveList editor window") and issuing other commands ("ROD[n] control/command window");
- to control and coordinate the execution of all commands for the ROD[n] and the "slave" threads.

...

3.3 PrimitiveListBuilding[n] thread

"Pulls out" **PrimitiveListTable** from **Control/status[n]** thread and according to its attributes "builds" PrimitiveList in local **InputMessageBuffer_host[n]**, which is the host mirror of **InputMessageBuffer[n]** allocated in **ROD[n]**.

(Note: **PrimitiveListTable** is a "fictitious" structure/database which contains all parameters necessary for PrimitiveList construction plus some additional attributes like expected output data format and destination for each primitive in the PrimitiveList - will be defined later).

Whether a **PrimitiveListTable** is available is distinguished by **primTableRdy[n]** signal issued by **Control/status[n]** thread.

For the synchronisation with **MessageTransferRoutine[n]** is used signal **primitiveListRdy[n]**.

Informative states: { IDLE, ACTIVE, LIST_READY, ERROR }

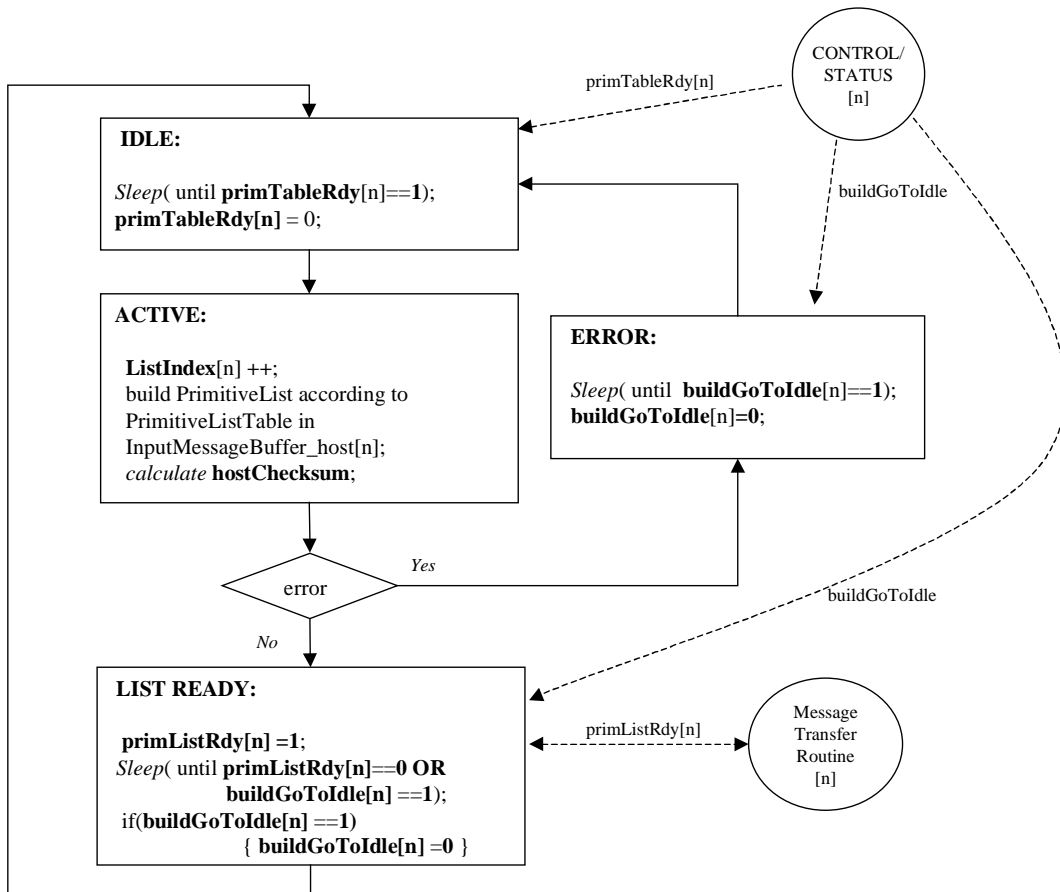
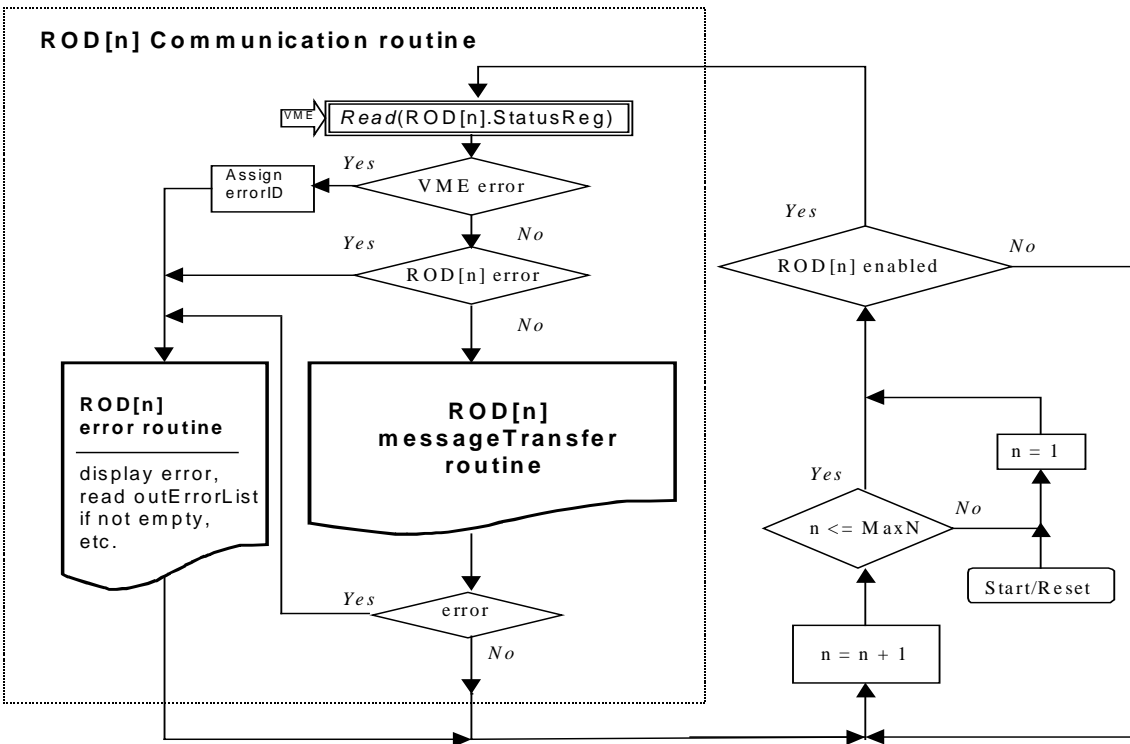


Fig. 2 PrimitiveListBuilding[n] thread diagram

3.4 CommonCommunicationLoop thread

This thread is common for all RODs in the crate.

Its purpose is to grant/distribute access to shared VME bus between all RODs in the crate on "round robin" basis.



Note: Readout of INFO, ERROR and DIAGNOSTIC BUFFERS not shown

Fig. 3 CommonCommunicationLoop thread(InfoBuffer, DebugBuffer and ErrorBuffer readout NOT INCLUDED)

3.5 MessageTransferRoutine[n]

This subset of **CommunicationRoutine[n]** is responsible for the data transfer between ROD[n] and Host through shared VME bus (plus handshaking and HPI settings), i.e. sending PrimitiveLists from **InputMessageBuffer_host[n]** to **InputMessageBuffer[n]** and OutputMessages from **OutputMessageBuffer[n]** to **OutputMessageBuffer_host[n]**.

The host side of "Communication Protocol" is implemented as a sequential state machine.

Very simplified state machine diagram showing the dependencies between the states:

```
messageTransfState { IDLE, POLL_DspAck_CLEAR, WRITE_PRIM_LIST,
    POLL_DspAck_SET, READ_OUT_MESSAGE, ERROR };
```

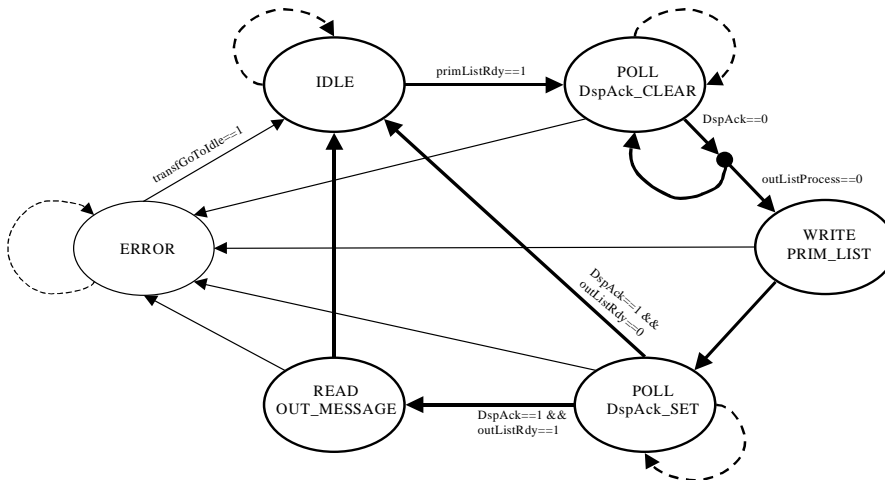


Fig. 4 MessageTransferRoutine - simplified state machine diagram

Since PrimitiveList and OutputMessage are always transferred as a whole, states WRITE_PRIM_LIST and READ_OUT_MESSAGE are only "transition" and serve mainly as a status information for the User.

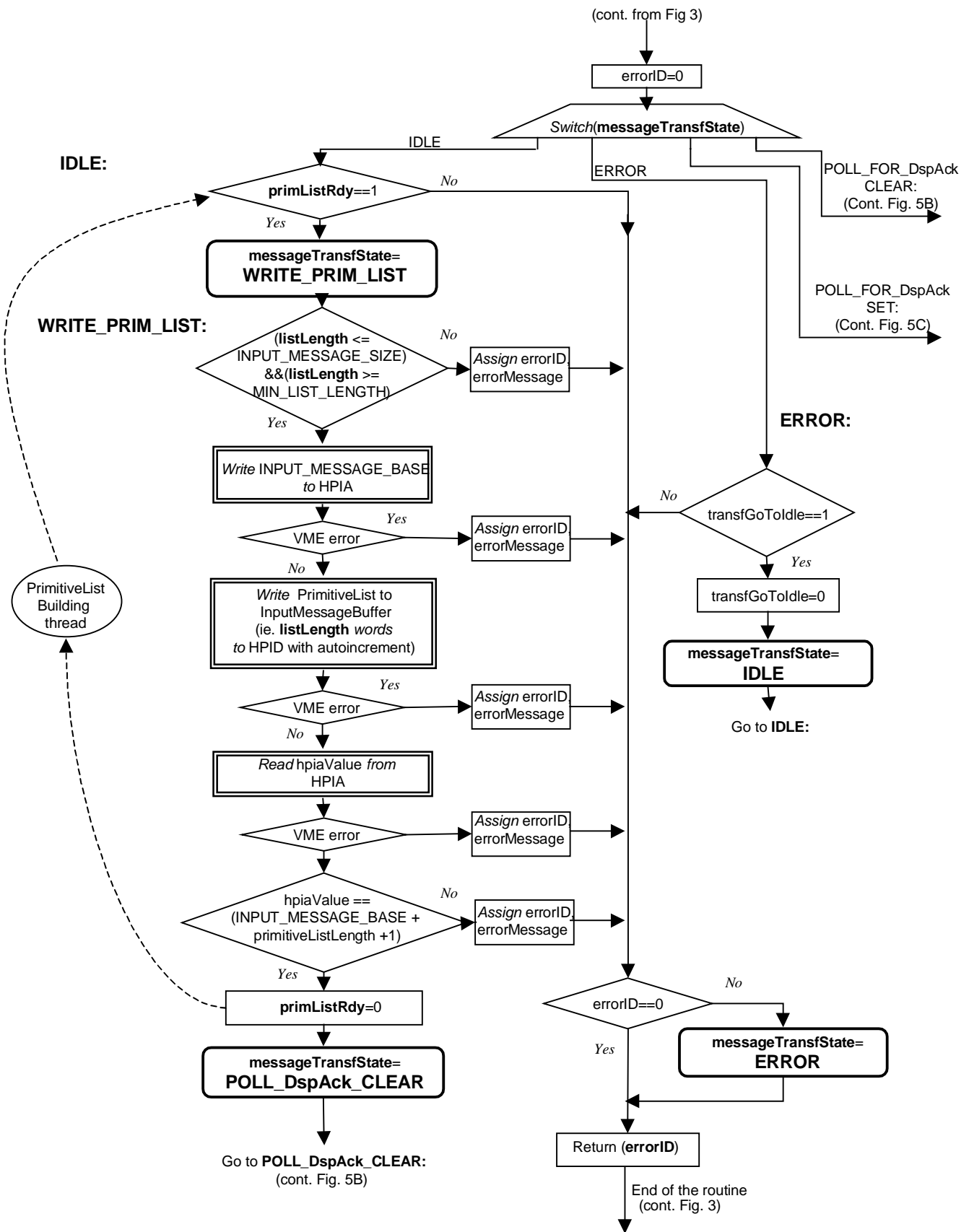


Fig. 5A MessageTransferRoutine - data flow diagram (IDLE , WRITE_PRIM_LIST and ERROR states)

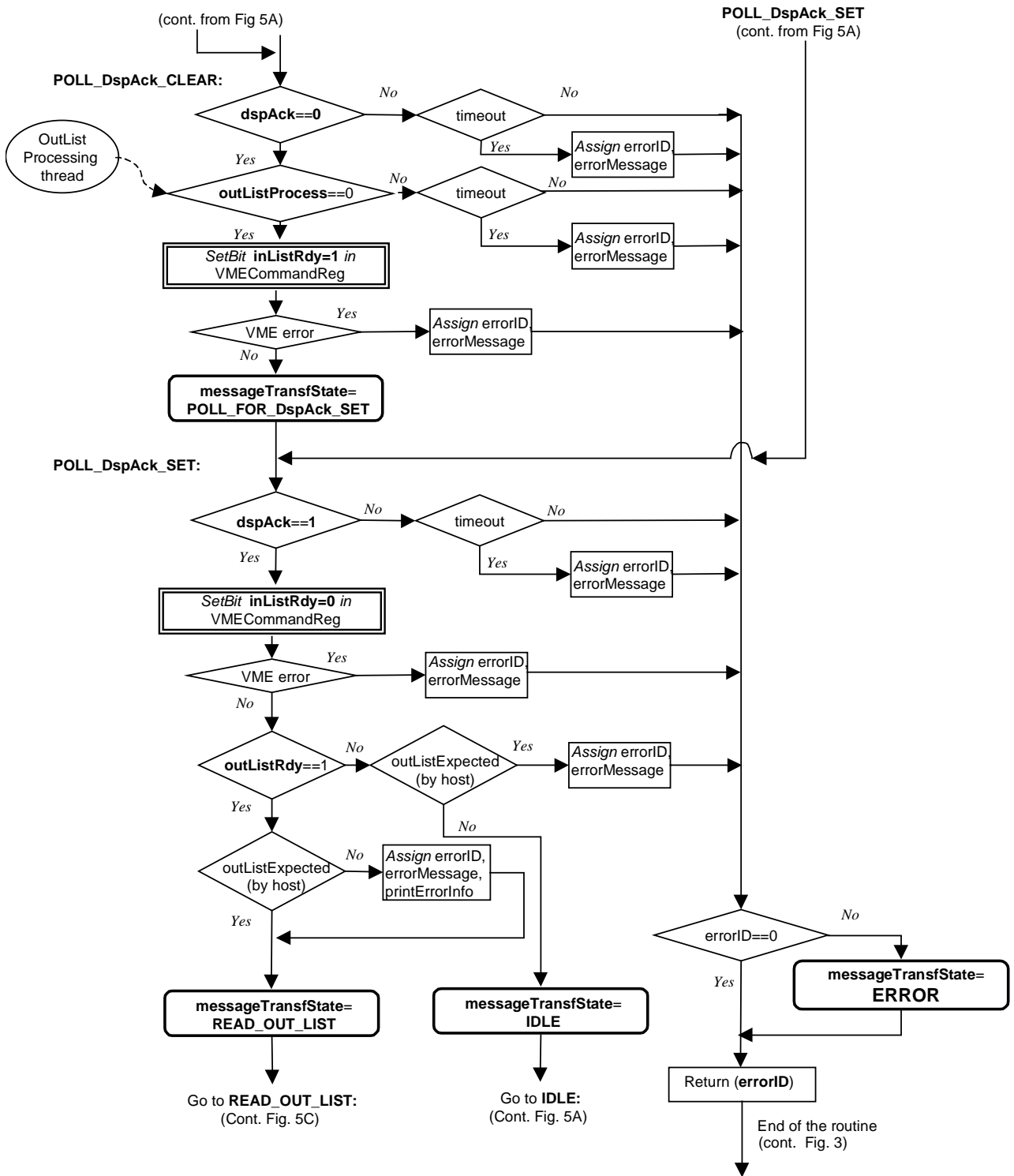


Fig. 5B MessageTransferRoutine - data flow diagram (POLL_DspAck_CLEAR , POLL_DspAck_SET states)

READ_OUT_MESSAGE: (Cont. from Fig. 5B)

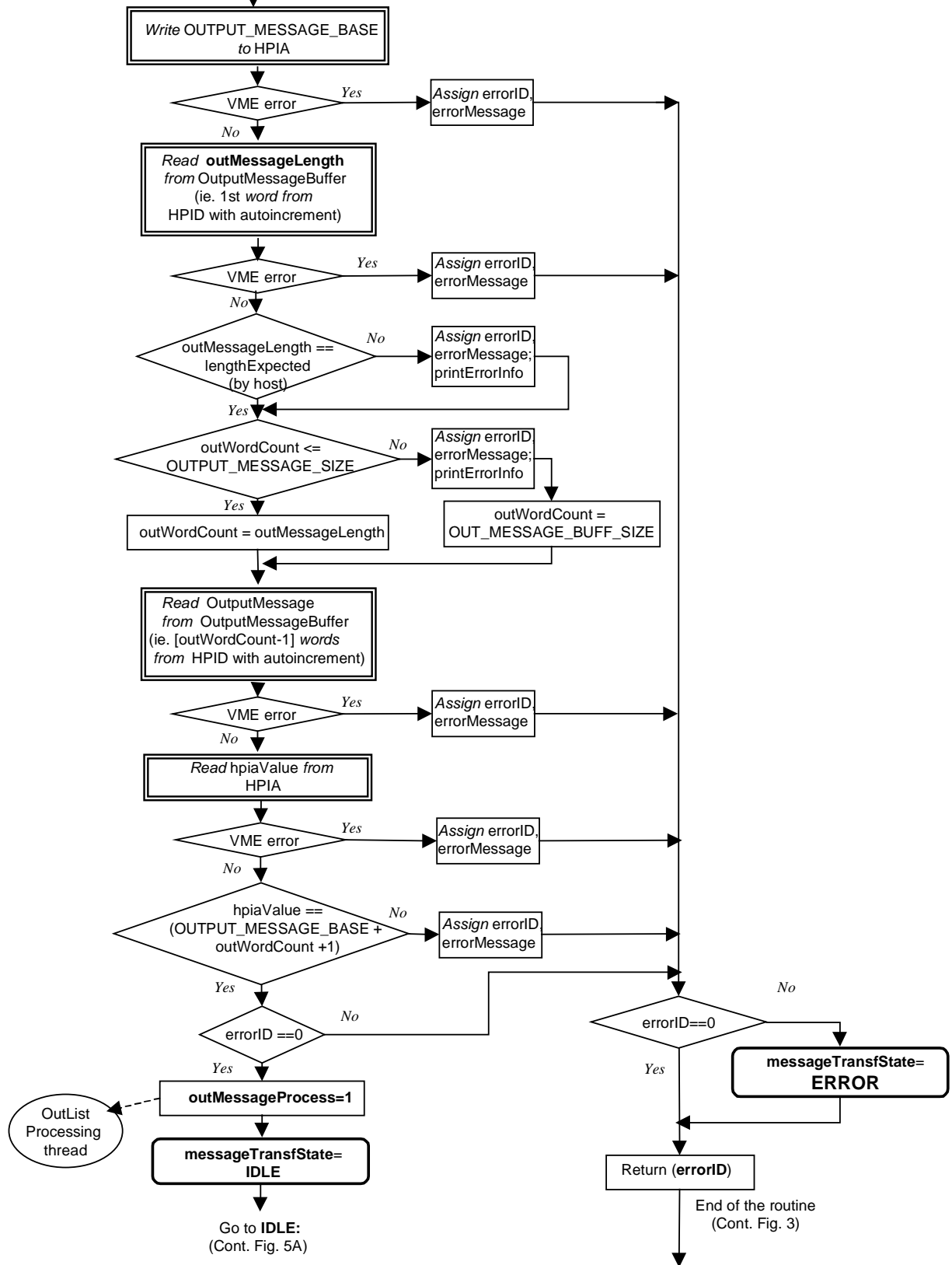


Fig. 5C MessageTransferRoutine - data flow diagram (READ_OUT_MESSAGE state)

3.6 OutputMessageProcessing[n] thread

This thread process outputMessage stored in OutputMessageBuffer_host[n], i.e. checks the validity of the data (checksum, length, consistency with PrimitiveList etc.) and then sends out the data to the predefined destinations (memory locations, files ...).

The thread gets active after it receives start signal **outMessageProcess** from MessageTransferRoutine.

When the data processing is over, **outMessageProcess** signal is cleared and Host can set **inListRdy** bit in VMCommandRegister if the next PrimitiveList has already been transferred to ROD.

Informative states: { IDLE, ACTIVE, ERROR }

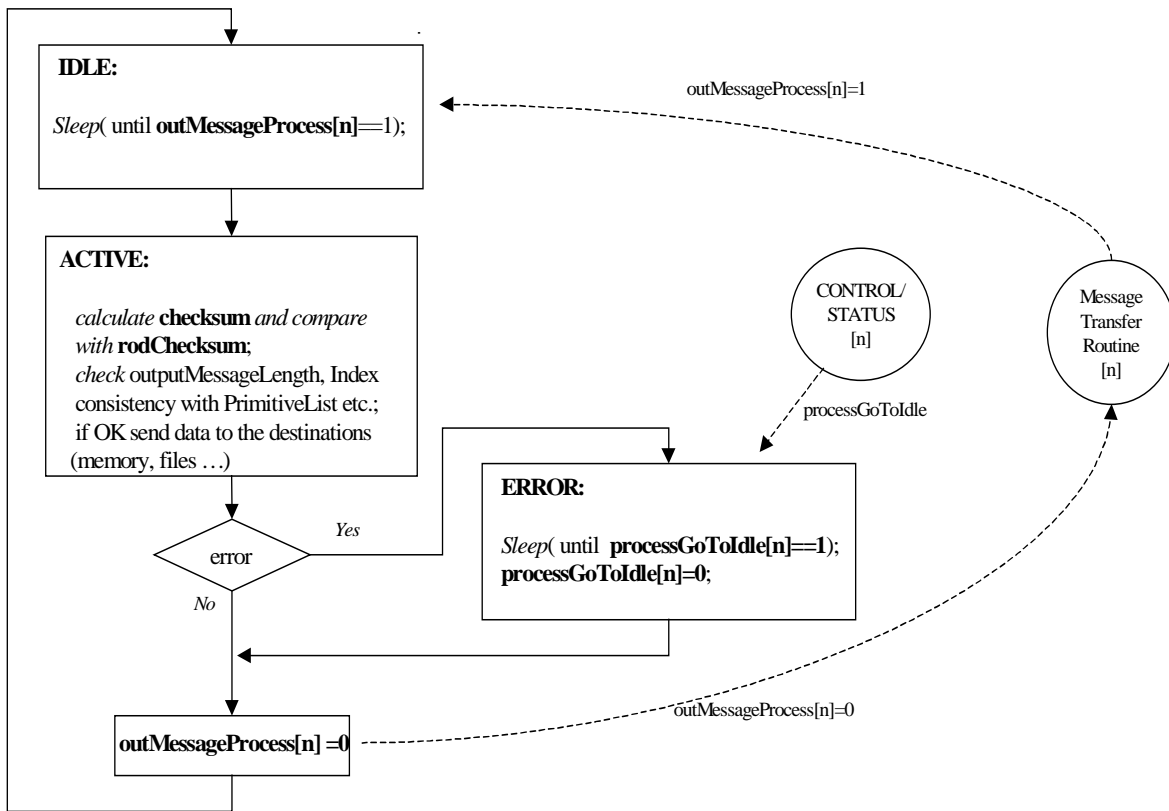


Fig. 6 OutputMessageProcessing thread diagram