Network Working Group
Request for Comments: 205
NIC: 7172

R. Braden
UCLA/CCN

6 August 1971

NETCRT - A CHARACTER DISPLAY PROTOCOL

At the May NWG, meeting, CCN circulated dittoed copies of a proposed character-display protocol NETCRT. Since that time, NETCRT has been revised significantly; the current version is now being published as an RFC, as promised last May.

NETCRT was developed because a particular site (RAND) requested Network access to URSA, CCN's display-based crje system. The primary use of URSA at UCLA is conversational remote job entry from a display terminal: entering and editing program text, submitting programs for batch execution, and examining job output; URSA is not a general-purpose time-sharing system.

URSA's text editor is designed for a fast updating character display and cannot be used in any reasonable way from a typewriter-like console. Therefore, a simple TELNET protocol is not adequate for using the crje function of URSA. Furthermore, we have assumed that other ARPA sites will have their own text editors, well matched to their own terminals and systems. Therefore, CCN has implemented NETRJS (see RFC #189), to provide remote job submission and retrieval services, before implementing NETCRT.

There are a number of other functions in URSA besides crje; some of these would probably be useful to remote users. URSA contains a comprehensive STATus service, whose constantly-updating displays are "windows" into the operation of the machine and the operating system, allowing a user to watch the progress of his jobs through the system. URSA also includes on-line data set (file) utilities, convenient for a user with files stored at CCN. To obtain access to these facilities, a few sites which use CCN heavily may want to implement NETCRT. The schedule for implementation of NETCRT at CCN to allow Network access to URSA will depend upon the existence of a user site that wants the service and that will write a suitable NETCRT user process. Interested sites are urged to contact the CCN Technical Liaison, Bob Braden.

Even though the implementation schedule for NETCRT is nebulous, we are publishing the specs now for several reasons. First, we would like comments and criticisms. Furthermore, NETCRT contains some features which may be useful in the protocol(s) now being developed for full graphical displays.

NETCRT PROTOCOL - VERSION 3

A. INTRODUCTION

The UCLA Campus Computing Network (CCN) node intends to provide Network access to its conversational remote job entry system URSA. The URSA system is display-oriented, supporting only character displays with local buffers (originally IBM 2260 displays, now CCI 301 TV display consoles). This document defines a third-level protocol called NETCRT which allows a Network user in a remote Host to look like a CCI console to URSA. NETCRT is defined in terms of a virtual character display ("VCD") terminal, simulated by a process in the user host.

URSA, like many on-line console systems, attempts to provide a good man/machine interaction by keeping tight control over the state of the terminal. On the other hand, the Network Working Group has deliberately built some "squishiness" into the standard Network protocols. We believe this squishiness is a conceptual mistake when dealing with remote man/machine interaction, and we would support protocol revisions to allow control over the effective communication compliance between processes in different hosts. However, this NETCRT protocol attempts to cope with the present squishiness, which is apparently built into a number of host's NCPs. In fact, we have arranged things so a host can improve response time and reduce Network traffic with NETCRT by using the message buffering inherent in his NCP.

B. THE VIRTUAL CHARACTER DISPLAY

A VCD consists of the following virtual hardware (see Figure 1):

1. A rectangular _display screen_ capable of displaying N lines of M characters.

2. A _local buffer_ of M x N characters used to refresh the display.

3. A _cursor register_ which addresses the characters in the buffer (and hence on the screen). This register controls the writing of text into the local buffer from either the keyboard or the server, and the reading of the local buffer by the server.

4. A _keyboard_ containing text keys and control keys. Each text key enters a character into local buffer at the current cursor address and steps the cursor register by 1.

5. A _communication interface_ through which the server CPU can send a stream of _command_ segments to the VCD and receive a stream of _response_ segments from the VCD. The command segments include control commands to the VCD and text to be written into the local buffer. Response segments contain status indicators and text read from the buffer. In addition, both VCD and server may send break signals.

The current address in the cursor register, an integer between 0 and M x N-1, is displayed as a blitch, underscore, or other visual indication at the corresponding point on the screen; this indication is called the _cursor_. Position 0 is the upper left corner of the screen.

The screen is addressed in line ("row") order, and read and write operations by the server overflow automatically from one line to the next. The cursor register is not assumed to operate modulo M x Nxsy. It is possible for a server command to set the cursor register to M x N, one position beyond the last screen position; however, the server should never set the register to an address beyond M x N, and it should not leave the cursor at M x N when the keyboard is unlocked.

The application program or conversational system using the VCD may format each display screen in a variety of ways, and may use a number of styles of interaction. One consequence is that the application program might have to look anywhere on the screen (i.e., in the local buffer) to find the input information it requires. We may consider three alternative mechanisms for transmitting information from the VCD to the serving CPU:

Mechanism 1 Whenever the user presses a "Transmit" control key, the entire M x N characters in the buffer are transmitted to the server CPU.

Mechanism 2 When the user presses "Transmit", the string of text between a "start" control character and the cursor is transmitted to the server.

Mechanism 3 The server must send a read command segment to the VCD before the "Transmit" key will have an effect. The read command segment determines which parts of the buffer are to be transmitted to the server.

Mechanism 1 may be faulted as too costly in transmission time and channel capacity, while Mechanism 2 is too restrictive. The scheme which we propose here is based on Mechanism 3, which subsumes the other two.

The VCD is assumed to include the following control keys:

Erase — Clears the display buffer to all blanks and resets the cursor to position 0 (the upper left corner of the screen).

Transmit — Locks the keyboard and places the VCD under control of the server CPU. Typically, the server will read specified areas of the screen and perhaps write out new data before unlocking the keyboard again.

Break — Has the same effect as _Transmit_, and in addition sends an interrupt message to the server CPU. The _Break_ key always sends the interrupt, regardless of the state of the VCD.

Reset — May be used to unlock the VCD keyboard in case the server CPU fails to respond immediately and the user wishes to enter new or different information.

These may be called pure control keys, since they do not correspond to any text characters. The following control key does store a character into the display buffer:

Newline — Enter a Newline (NL) character into the display buffer and reset the cursor to the beginning of the next line. If this character is encountered during a read or write operation, it is executed (i.e., the cursor is moved to the beginning of the next line) and the NL is counted as _one_ character.

Finally, there are assumed to be keys for manually positioning the cursor to any address on the screen. Cursor positioning keys may include: cursor right, cursor up, cursor left (BS), cursor down (LF), and cursor return (CR). A tab (HT) mechanism could also be defined, although none is included here.

C. VCD STATES

The VCD has two internal states, _Local_ and _Control_ (see Figure 2).

Local State: The keyboard is unlocked and all keys are active. The VCD does not accept or recognize any commands from server except (reverse) Break.

Control State: The keyboard is locked, and only the _Break_ and _Reset_ keys are active. The VCD accepts and executes command segments from the server, and returns response segments as the result of read commands.

The VCD changes from Local to Control state if either:

(1) The user presses the _Transmit_ key; or

(2) the user presses the _Break_ key; or

(3) the server sends a reverse Break request.

Transmit's only effect is to enter Control State; _Break_ enters Control State and also sends a break request (INS and X'80') to the server.

The VCD returns to Local State when either:

1. The user presses the _Reset_ key; or

2. the VCD encounters a LOCAL command from the server and is not in the process of synchronizing a reverse break (see section E below).

We should note that CCI and IBM 2260 character display consoles actually have only one control key ("Interrupt" on CCI, "Enter" on 2260) to perform the functions of both _Break_ and _Transmit_; this one key in fact has the function of the _Break_ key of the VCD. We have included both _Break_ and _Transmit_ keys in the VCD for generality, but the URSA-NETCRT interface will be programmed to allow a Network user of URSA to either (1) employ the _Break_ key exclusively, or (2) use either _Break_ or _Transmit_ as appropriate. This will be achieved by URSA simply by ignoring those break requests (INS messages) which occur while there are outstanding read commands.

D. VCD COMMANDS

The server sends the VCD a string of command segments. These are of varying length, consisting of an op code and none or more parameters. The commands recognized by the VCD are as follows:

1. Display & Keyboard Control Commands:

Command	Parameter(s)	Function
ERASE	none	Erase display and reset cursor to 0. i.e, clear the local buffer.
BLANK	none	Disable display refresh (i.e., blank the screen but do not clear the local buffer).
UNBLANK	none	Enable display refresh.
LOCAL	none	Put VCD in _local_ state. The result is to suspend command interpretation and unlock the keyboard.
SYNC	none	Used to synchronize reverse Break from server. SYNC (X'80) is placed in stream by server at same time that it sends an INS. VCD enters Control State, synchronizes INS with BREAK command (see next section), and continues command interpretation.

2. Cursor Control Commands:

CURSOR	16 bit integer P	Set cursor register to P, where 0 <= P <= M x N.
FIND	X'0001' followed by one character c	Move the cursor to point to an occurrence of the character c. Specifically, search backwards toward lower addresses) from the current cursor position and take the first occurrences of c (i.e., the one with the largest address). If no occurrence is found, leave cursor at position 0.
SAVE	none	Save a copy of the current cursor address in local register S.
RESTORE	none	Replace cursor register contents by value S.

I/O Commands:

WRITE n,text	16 bit integer n, followed by n text bytes.	Write n bytes of text into display buffer starting at current cursor position and advancing cursor by 1 for each byte (except NL character advances to beginning of next line). Here [sigma] + n <= M x N.
READ n	16 bit integer n.	Read n bytes starting at the cursor [sigma] and advancing cursor by one for each byte (except NL advances cursor to beginning of next line). NL counts as one character. Send the text to the server as a response segment. Must have n + [sigma] <= M x N.
SREAD	none	Read S - [sigma] bytes starting from the current cursor position [sigma] up to (but not including) the cursor address stored in register S. The cursor is left in position S as a result. Send the text to the server as a response segment.
AWRITE n,text	16 bit integer n, followed by n text bytes.	Same as WRITE n, except characters are not stored in buffer if they have a lower cursor address than the value in S.

Here are some applications of these commands in URSA:

1. One elementary URSA terminal operation reads the screen from position x up to (but not including) the current cursor position. This could be done with the sequence of VCD command segments:

```
SAVE
CURSOR x
SREAD
```

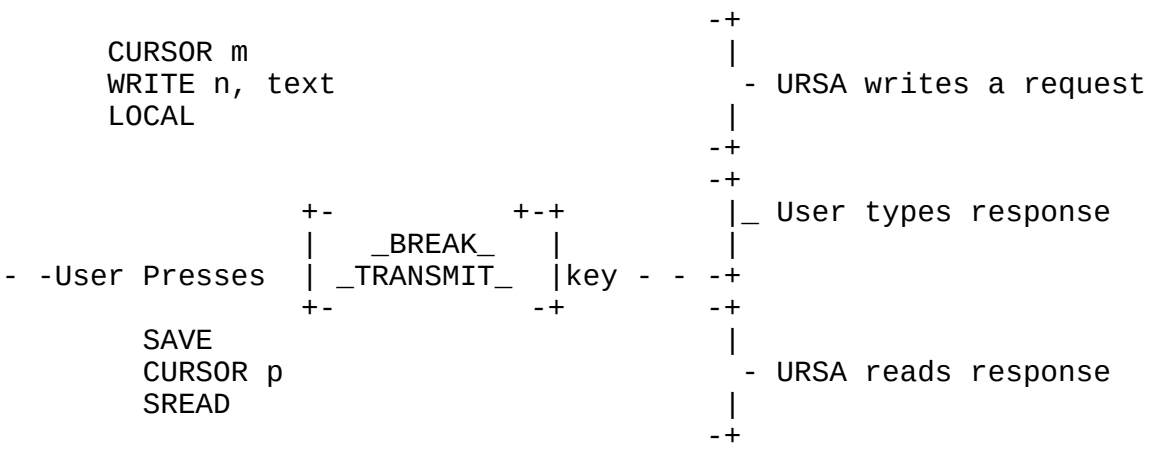
2. Another common operation in URSA is to remember the cursor, update specific information on the screen, and replace the cursor. This can be done by the following 8 + n byte sequence of command segments:

```
SAVE
CURSOR x
WRITE n, text
RESTORE
```

3. In URSA, the area in which a user is to type his response is usually delimited on the left by a "Start Symbol" (graphic '[1]'). This is a historical remnant of the IBM 2260, which has only two hardware read operators: read the full screen, and read from the Start Manual Input Symbol ("SMI") to the cursor. The SMI read operation can be simulated easily on the VCD as follows:

```
SAVE
FIND '[1]'
SREAD
```

4. The _Break_ (or _Transmit_) key on the VCD may serve the function of the INTerrupt key on a CCI console (or ENTer on an IBM 2260). URSA will often attempt to minimize Network traffic by sending a sequence of commands (one message if allocation allows) like the following:



At other times, URSA might send the sequence:

```
CURSOR m
WRITE n,TEXT
LOCAL
READ  0
```

and wait for the INS from the user pressing _Break_ (or the response segment triggered by the zero-length read if he presses _Transmit_); then URSA will send the appropriate read command sequence.

F. NETWORK MESSAGE FORMATS

The VCD connects the server through ICP to a standard socket, establishing thereby a pair of connections between the VCD and the server. Command segments (server-to-VCD) and response segments (VCD-to-server) are sent over these connections, without regard to physical message boundaries, using byte size 8. The VCD is defined to operate in a segment-at-a-time mode (rather than character-at-a-time), with local echo. Therefore, the server never echoes under NETCRT.

In many cases URSA will send a sequence of command segments (as in the examples of the preceding section) at once; if there is sufficient allcocation they will be sent in the same message. Response time may be improved, therefore, if the user site is able to buffer ahead on command segments. This buffering does raise break synchronization problems, which are solved in the following manner for reverse (server-to-user) break:

> The server sends an INS on the control link and also a SYNC command (X’80) on the data link to the VCD. On receiving either, the VCD enters Control State and then achieves synchronization between the INS and BREAK; if the INS arrives first, the VCD executes normally all commands buffered in his host, _except_ it ignores LOCAL commands, until the SYNC appears. Having achieved synchronization, the VCD continues normal command interpretation (without ignoring ensuring LOCAL commands).

By this means the server can regain control of the VCD to write new information at any time. For example, when URSA is used under NETCRT, most WRITE or AWRITE sequences will be preceded by a BREAK from the server, since URSA will not know the current state of the VCD. Even if URSA left the VCD in Control State, the user might have manually returned his VCD to Local State by pressing _Reset_.

After receiving an INS, the VCD executes rather than ignores buffered commands so that pending writes will not be lost in case that processing at the user side has been held up temporarily. The read commands executed after the server sent an INS might be irrelevant to a server, which can ignore the corresponding response segments. In order to do so, the server simply keeps matching counts of read commands sent and corresponding response segments received.

Command segments will use the following formats:

Form 1 (No parameters) q:OPCODE(8)

> where q = X’80’ means SYNC

X’91’ " LOCAL

X’92’ " ERASE

X’93’ " BLANK

X’94’ " UNBLANK

X’95’ " SAVE

X’96’ " RESTORE

X’97’ " SREAD

Form 2 (16 bit integer) q:OPCODE(8) + n:INTEGER(16)

where q = X’9E’ means READ n

q = X’9C’ " CURSOR n

In both cases, 0 <= n <= M x N

Form 3 (count and text) q:OPCODE(8) + n:LENGTH(16) + (TEXT(8) = n)

where q = X’9D’ means WRITE

q = X’9A’ means AWRITE

q = X’9F’ and n=1 means FIND

A response segment, caused by a READ or SREAD command, has the following format:

RESPONSE <-----X’A1’ + CURSOR(16) + n:LENGTH(16) + (TEXT(8) = n)

where n > 0 is the number of characters actually read. CURSOR(16) is an integer giving the final cursor position after the corresponding read command. Note that the command READ 0 is permissible and may be used by the server to find the current cursor position, or to find out when the user presses _Transmit_.

E. SCREEN SIZE

For simplicity and consistency with URSA, we have chosen to treat the cursor as a single integer. This in turn means that VCD and server must agree upon the number of columns M; it is also desirable for the server to know N.

The agreement on M and N takes place through a one-sided negotiation. The server is assumed to know what M and N values he can handle and these are published for user sites. When the VCD is first connected to the server, the VCD must send an Open response segment with the values M and N:

Open segment <---- X'B1' + M(8) + N(8) + X'0000'

If the VCD fails to send this segment or the server does not like the values, the server closes the connections and the user is considered logged off.

Endnotes

[1] Graphic representation of start symbol: shaded triangle on its side.

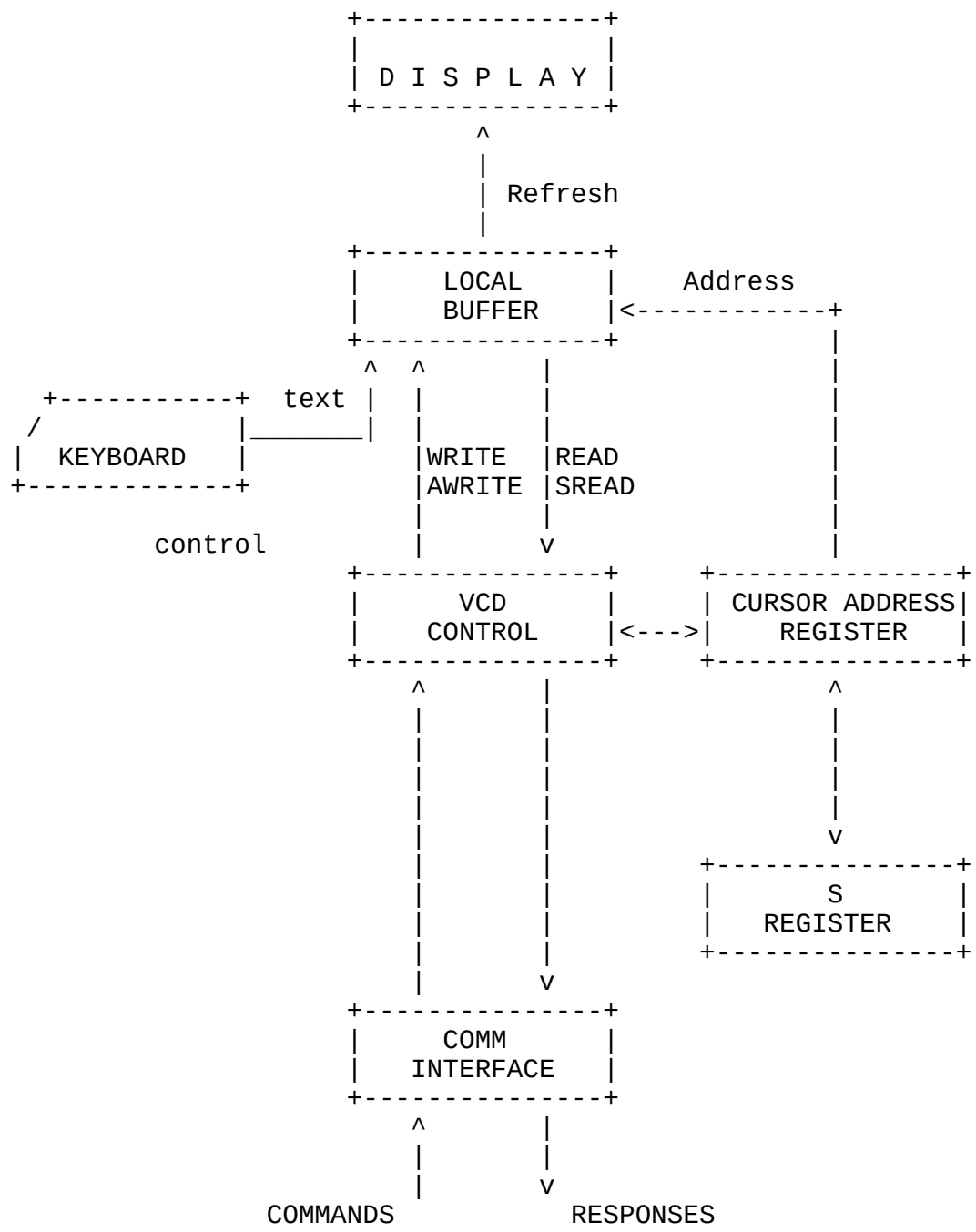


FIGURE 1. VIRTUAL CHARACTER DISPLAY

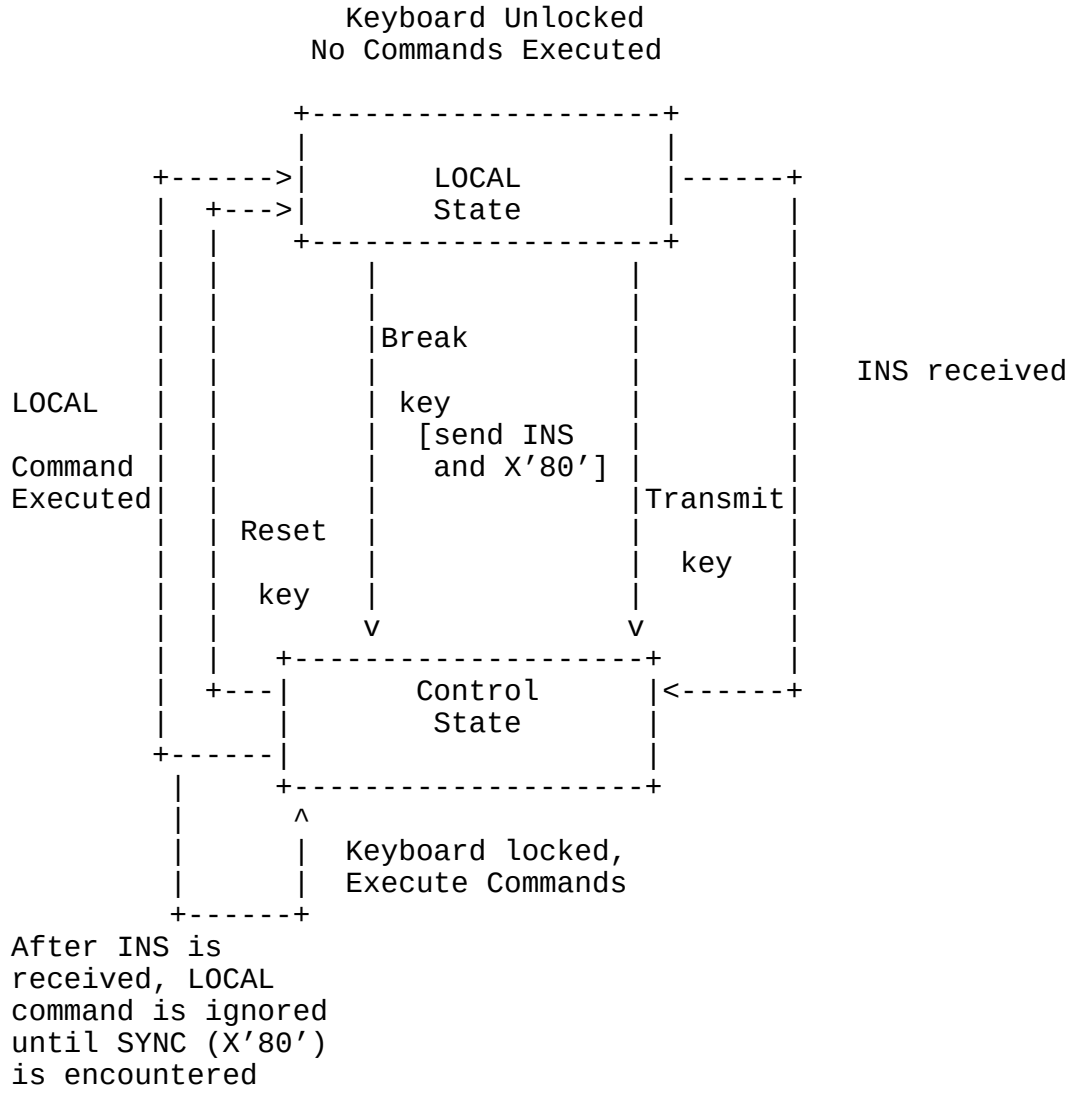


FIGURE 2. VCD STATES

[This RFC was put into machine readable form for entry]
[into the online RFC archives by Lorrie Shiota, 2/02]